Comparative Performance Evaluation of Garbage Collection Algorithms

Benjamin G. Zorn

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

December 1989

Comparative Performance Evaluation of Garbage Collection Algorithms

This research was funded by DARPA contract numbers N00039-85-C-0269 (SPUR) and N00039-84-C-0089 (XCS) and by an NSF Presidential Young Investigator award to Paul N. Hilfinger. Additional funding came from the Lockheed Corporation in the form of a Lockheed Leadership Fellowship and from a National Science Foundation Graduate Fellowship.

Abstract

This thesis shows that object-level, trace-driven simulation can facilitate evaluation of language runtime systems and reaches new conclusions about the relative performance of important garbage collection algorithms. In particular, I reach the unexpected conclusion that mark-and-sweep garbage collection, when augmented with generations, shows comparable CPU performance and much better reference locality than the more widely used copying algorithms.

In the past, evaluation of garbage collection algorithms has been limited by the high cost of implementing the algorithms. Substantially different algorithms have rarely been compared in a systematic way. With the availability of high-performance, low-cost workstations, trace-driven performance evaluation of these algorithms is now economical. This thesis describes MARS, a runtime system simulator that is driven by operations on program objects, and not memory addresses. MARS has been attached to a commercial Common Lisp system and eight large Lisp applications are used in the thesis as test programs.

To illustrate the advantages of the object-level tracing technique used by MARS, this thesis compares the relative performance of stop-and-copy, incremental, and mark-and-sweep collection algorithms, all organized with multiple generations. The comparative evaluation is based on several metrics: CPU overhead, reference locality, and interactive availability. Mark-and-sweep collection shows slightly higher CPU overhead than stop-and-copy collection (5%), but requires significantly less physical memory to achieve the same page fault rate (30–40%). Incremental collection has very good interactive availability, but implementing the read barrier on stock hardware incurs a substantial CPU overhead (30–60%). In the future, I will use MARS to investigate other performance aspects of sophisticated runtime systems.

Acknowledgments

There are many people that deserve my heartfelt thanks. I only regret that I cannot mention them all by name here.

First, I want to thank Paul Hilfinger, my research advisor, who with care and diligence, helped me understand the nature and substance of quality research and guided my research along that path. His comments improved the initial drafts on this work considerably.

I'd also like to thank the other readers, Richard Fateman and Phil Colella. By providing a different perspective on the writing, they helped me make the contents more accessible to all readers.

This thesis is a direct result of my involvement with the SPUR project and in particular, my participation in the design and implementation of SPUR Lisp. I'd like to thank Jim Larus for building the SPUR Lisp compiler and for his help with the design of the runtime system. I'd like to thank Kinson Ho for his comments on the thesis text, which were especially relevant coming from someone who was quite familiar with aspects of the implementation discussed.

Part of efficient garbage collector design involves an intimate understanding of computer architecture. George Taylor, one of the designers of the SPUR architecture, was an invaluable resource to me. By working closely with him on the SPUR simulator, I learned more things about the SPUR architecture than I thought I'd ever want to know. Shing-Ip Kong, another SPUR architect, helped me understand the inherent tradeoffs in architecture design.

Garbage collection is all about memory management, and any algorithm must be written with an understanding of the underlying memory system provided by the hardware. I'd like to thank David Wood and Susan Eggers for helping me whenever I needed to know more about caches. Garth Gibson helped me by answering questions about the cache/main memory interactions. Finally, I am in debt to Mark Hill for allowing me to use his Dinero and Tycho cache simulation tools.

Between the program runtime system and the hardware lies the operating system. Members of the Sprite operating system team helped me understand the kinds of OS support the runtime system should count on. Brent Welsh, Mary Gray Baker, and John Ousterhout provided me with information about Sprite protection faults. Mike Nelson helped me understand the Sprite virtual memory system. I'd also like to thank Fred Douglis and Andrew Cherenson for their help with my questions.

I have also received help from people outside the university. I am indebted to Doug

Johnson of Texas Instruments for the benefit of his wide-ranging knowledge about Lisp applications and implementations, and his specific knowledge of Lisp machine garbage collection. I am greatly indebted to all the people at Franz, Inc. for letting me use their Common Lisp system in my research, and for giving me valuable feedback about specific aspects of my thesis.

The preparation of a thesis is a complicated process requiring knowledge of such diverse tools as awk, csh, latex, S, postscript, idraw, gremlin, and make. I'd like to thank Luigi Semenzato for explaining some of the more subtle interactions between these tools to me. I'd also like to thank Ken Rimey and Edward Wang, for their inspiring discussions and their large Common Lisp programs. Dain Samples was helpful by providing me with mache, a tool for greatly compressing simulation trace files.

Finally, I'd like to thank my parents, both for reviewing parts of the text, and for giving me the education and motivation to write this thesis in the first place.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Garbage collection is a technique for automatic reclamation of allocated program storage first suggested by McCarthy [60]. Other techniques for storage reclamation exist, most notably explicit programmer-controlled reuse of storage (used in Pascal, C, etc.) and reference counting [23]. In this thesis, the term *garbage collection* refers to the automatic storage reclamation technique in which storage is reclaimed by a periodic traversal of the program objects to identify those that can be used further and those that cannot. Objects in this latter category are referred to as *garbage*. The non-garbage objects are identified by starting from a special set of object references (the *root set*) and transitively traversing all program objects reachable from the root set. The methods for traversing objects and noting them as reachable (preserving them) differ widely between algorithms, but all garbage collection algorithms operate with the same basic goal of periodically traversing and re-partitioning storage.

1.1 Automatic vs Explicit Storage Reclamation

Many languages require the programmer to explicitly release objects that are no longer in use (e.g. the `free` function in C and Pascal). Explicit programmer management of storage leads to two very common and insidious bugs: memory leaks and duplicate releases. A memory leak occurs when objects are allocated but then not released when they are no longer necessary. Eventually, such a leak can cause a long running program to consume the entire virtual address space and fail. Even if the program does not fail due to the memory leak, the locality of reference can suffer greatly as the objects in use get spread across a large virtual address space. Although tools exist to help detect memory leaks [8, 95], leaks are easy to create and difficult to locate. Programmers may have a hard time correctly identifying when an object is no longer in use because such a decision requires global information about the program data, and in large systems with many implementors, this global information may be unavailable.

If a programmer uses `free` carelessly, he can cause the other insidious bug, the duplicate release, which occurs when an object that is still reachable from other program objects is released. Such a release causes an inconsistency when the released object is reallocated because the same memory location is then being used to represent two distinct objects. This inconsistency is difficult to detect because the program may fail long after the inconsistency is created and the connection between the duplicate release and the program failure may not be obvious.

The problems involved in explicitly managing storage have caused language designers to redesign languages to include automatic storage reclamation. For the language Mesa it was estimated that developers spent 40% of the development time implementing memory management procedures and finding bugs related to explicit storage reclamation [71]. Cedar, the language defined as a successor to Mesa, includes automatic storage reclamation. Likewise, Modula-3, a successor to Modula-2 defined at DEC Western Research Lab, incorporates automatic storage reclamation [16]. Many other modern languages incorporate automatic storage reclamation including CLU [56], Prolog [20], Smalltalk [37], and of course, Common Lisp [76]. Efforts are even being made to incorporate "conservative garbage collection" in C programs.

1.2 Garbage Collection and Reference Counting

Reference counting is another widely used technique for automatic storage reclamation. In this approach, a count is stored with each object recording the number of references to the object. When a copy of the object reference is made, the count is incremented. When a reference to the object is destroyed, the count is decremented. When the count reaches zero, no more references to the object exist and the object can be reclaimed. Reference counting has the advantage that storage is reclaimed incrementally as references to objects disappear. Furthermore, Deutsch and Bobrow introduced a deferred reference counting technique that reduces the distributed overhead of maintaining the count [27].

Unfortunately, reference counting has fundamental disadvantages. First and foremost, reference counting algorithms do not reclaim storage allocated in circular structures. Modifications to the traditional algorithm have been suggested to overcome this problem [15], but the performance of the modified algorithm is unacceptably slow. As a result, reference counting algorithms are augmented with traditional garbage collection algorithms. A second disadvantage of reference counting is that space is required to maintain the count. A simple implementation associates a 32-bit count with each object and increases the size of each cons object (the most common object type) by 50%. More complex implementations reduce this overhead but do not entirely eliminate it. A third disadvantage of reference counting is that it fails to reorganize or compact objects in memory and is thus unable to improve the locality of references to those objects. By using generations, garbage collection can significantly improve reference locality, as this thesis shows. Finally, the most significant advantage of reference counting, that of incrementally collecting storage, has also been achieved with garbage collection algorithms using incremental [6] and generation [55] techniques.

Because of these disadvantages, reference counting is not often used in modern Lisp and Smalltalk implementations. Recently, however, research with distributed memory computers has sparked renewed interest in reference counting algorithms because they allow storage deallocation based on local information, instead of the global information required by garbage collection [10, 38]. This dissertation focuses entirely on techniques for garbage collection and does not consider reference counting any further.

1.3 Techniques for Garbage Collection

Having established that other storage reclamation techniques have serious drawbacks, I now discuss the need for comparative performance evaluation of garbage collection algorithms. While all collection algorithms incorporate the basic ideas of traversing and preserving reachable objects, the implementation of these ideas has changed dramatically since the first algorithm was proposed in 1960 by John McCarthy [60]. Effective algorithms have necessarily adapted to changes in the hardware and software systems in which they are implemented. Historically, algorithms have traditionally remained effective only over a period of a few years because of rapid changes in technology. Looking at the development of algorithms from a historical perspective, we can see past trends and understand the need for effective evaluation techniques to provide answers about the effectiveness of algorithms in future systems.

1.3.1 1960's — Mark-and-Sweep Collection

The original algorithm proposed by McCarthy was a simple mark-and-sweep collection. As the name implies, collection is divided into two phases: the mark phase, in which reachable objects are traversed and marked, and the sweep phase, in which the memory is scanned and garbage objects are collected for reuse. An optional phase, called the compaction phase, relocates objects so they are packed close together in memory. Marking is accomplished by reserving a bit in each object to indicate that it has been marked. In traversing reachable objects, a mark stack is usually necessary to allow the algorithm to backtrack and follow each pointer contained in an object.

The systems for which mark-and-sweep collection was originally designed contained small (by current standards) physically addressed memories. At the time, the execution cost of scanning the entire memory was negligible. Of more direct importance was the need to preserve space in the memory for the mark stack, which in principle requires space proportional to the size of the entire memory. Schorr and Waite developed a clever algorithm that avoided the need for the mark stack by reversing pointers during the traversal [72].

Mark-and-sweep collection has distinct drawbacks that become apparent when considering larger virtual memories. Its primary drawback is that collection time (sweeping in particular) is proportional to the size of the memory. As memory sizes increase, garbage collection overhead increases proportionally. A second disadvantage of mark-and-sweep collection is that variable-sized objects (e.g., vectors and strings) require special handling during collection. Either objects must be compacted to squeeze out holes between objects or the holes reclaimed by collection must be allocated with a policy such as first-fit or best-fit that could result in loss of memory to fragmentation. Although neither of these disadvantages is insurmountable, a new technique for collection was introduced in the early 1970's and it gained wide acceptance.

1.3.2 Early 1970's — Copying Garbage Collection

In 1969, Fenichel and Yochelson suggested the first copying collection algorithm [30]. In 1970, Cheney suggested a modification to their algorithm that avoided the need for an auxiliary stack [18]. The so-called Fenichel-Yochelson-Cheney (FYC) copying algorithm has remained the basis for effective garbage collection algorithms ever since. Fenichel and Yochelson suggested dividing the program memory into two *semispaces*. Objects are allocated in one semispace until a garbage collection is required (i.e., all the space has been allocated). During garbage collection, as objects are traversed, they are copied to the other semispace. In addition, a *forwarding pointer* is left behind so that additional references to the object can be updated correctly. When garbage collection is over, all reachable objects are now located in the other semispace and allocations now take place in that semispace. Thus the "from" and "to" semispaces change roles after a garbage collection "flip." Cheney improved this algorithm by indicating how the transported objects could also serve as the stack of objects being traversed.

The FYC copying algorithm traverses, preserves, and compacts objects in time proportional to the number of reachable objects and not the size of the memory. The term "garbage collection" is somewhat of a misnomer in this case, since what happens is really non-garbage compaction.

The biggest cost of copying garbage collection is that half of the address space is unavailable for allocation of objects. A major reason for the success of copying garbage collection was that virtual memory increased the size of address spaces so that such a flagrant loss of address space was acceptable. Even though copying garbage collection requires a larger address space, the inherent compaction it performs can actually provide better locality of reference than a non-compacting mark-and-sweep algorithm.

By the late 1970's, systems technology had changed enough that new algorithms for garbage collection were required. Computers became faster, memories became larger, and Lisp programs, such as MACSYMA[1], became significantly larger. As program data increased from tens of kilobytes to tens of megabytes, the time required to collect the data increased. Since Lisp is part of an interactive programming environment, fast response time

[1]MACSYMA is a symbolic algebra program written in Maclisp. The source of MACSYMA contains over 100,000 lines of Lisp.

is important to its users. By the late 1970's pauses resulting from garbage collection could last tens of seconds or more.

1.3.3 Late 1970's — Baker Incremental Garbage Collection

In 1978, Henry Baker at MIT suggested a technique that solved two important problems caused by garbage-collection-induced pauses [6]. First, Lisp was being used for AI programming and garbage collection interfered with real-time uses of Lisp for applications such as robot control. Second, Lisp was an interactive language and increasingly long garbage collection pauses disrupted interactive users.

To solve these problems, Baker introduced a technique he called incremental garbage collection to augment the traditional FYC copying algorithm. Baker's idea was to avoid copying all objects at flip time and instead incrementally copy objects from the other semispace as new objects are allocated. This is possible by maintaining the illusion that objects have all been copied at the time of the flip. This illusion is maintained through implementing what is called the *read barrier*. Whenever a pointer is loaded from memory, it is first checked to determine if it points into the semispace being copied from (*fromspace*). If so, the object being referenced is immediately transported (copied) to the current semispace (*tospace*) and the reference to it is updated. This action maintains the invariant that all pointers in machine registers point to objects in tospace. To guarantee that all objects in fromspace are eventually transported to tospace, whenever an object is allocated some objects are transported into tospace as well. By associating transporting with allocation, the garbage collector can guarantee that tospace will not be exhausted before all objects have been copied from fromspace. Because only a small number of objects are copied at each flip (when fromspace and tospace change roles), there is no perceptible pause associated with garbage collection.

In analyzing his algorithm, Baker concludes that the execution cost is nearly identical to the cost of traditional copying garbage collection because the same objects are transported, except that they are transported at different times and in different orders. The additional overhead of incremental garbage collection lies in maintaining the read barrier. With hardware support like that provided by the Symbolics and TI Explorer Lisp machines, the overhead of maintaining the read barrier is negligible and incremental collection has been effectively implemented [25, 61]. However, without hardware support, implementing the read barrier can be very costly since a large fraction of memory references are loads. Hybrid forms of incremental collection have been implemented successfully on stock hardware [29], but Baker incremental collection has never been implemented successfully without hardware support.

While incremental garbage collection solves problems associated with long garbage collection pauses, it is not necessarily an appropriate solution for general purpose computers, and it fails to solve other problems associated with large Lisp systems. In particular, by the early 1980's Lisp systems had become so large they could not fit comfortably in the available physical memories.

1.3.4 1980's — Generation Garbage Collection

In the early 1980's, work with large Lisp programs, such as MACSYMA, led to suggestions for new garbage collection techniques [32]. Because the Lisp images were larger than physical memories, garbage collection could cause intense thrashing of the virtual memory system. In 1983, Lieberman and Hewitt proposed generation garbage collection, in which a small part of the heap is collected independently of the rest of the heap [55].

The main purpose of generation garbage collection is to focus the effort of collection on a subset of the total heap. Empirical measurements show that most objects become garbage shortly after they are created, and so the most effective subset of the heap to collect is the youngest [74]. Lieberman and Hewitt divided the memory into areas by age and called these areas *generations*. Generations containing the youngest objects are collected frequently. Generations containing older objects are rarely collected. Objects that survive long enough in a particular generation are *promoted* to an older generation and thus are collected less frequently.

Throughout this thesis, the youngest generation is referred to as *newspace* and the older generations are collectively referred to as *oldspace*. In this text, oldspace and newspace are never used to distinguish between the semispaces of a copying collector, as they sometimes are in the literature. The terms fromspace and tospace are used exclusively for that purpose. Newspace is also called the *first generation* and the next oldest generation (to which newspace objects are promoted) is called the *second generation*.

While Lieberman and Hewitt specified a particular process for collecting and promoting, the basic idea of generation garbage collection has been adapted and applied in different forms. Ungar showed that a very simple variant which he called generation scavenging could be effective in Smalltalk [84]. Moon demonstrated a variant called ephemeral garbage collection on the Symbolics [61]. Variants of generation garbage collection have also been implemented on stock hardware [33, 75].

Being able to collect a single generation requires that all reachable objects in the generation are identifiable. In particular the traditional root set, which would normally include the machine registers and runtime stack, must be augmented to include pointers stored in other generations that point into the generation being collected. To allow collection of any generation independently of another, a generation algorithm needs to record all intergenerational pointers. In practice, if the algorithm collects a generation and all younger generations at the same time, then it only needs to record pointers forward in time—that is, pointers stored in older generations that point into younger generations.

Generation garbage collection has been very successful in improving the locality of reference of garbage collection in large Lisp systems. In addition, because the generations being collected are relatively small, collection times for a generation can be short and potentially non-disruptive to interactive users. Nevertheless, there are additional costs for generation collection. Maintaining the locations of pointers forward in time requires special checks when pointers are stored. These checks are sometimes referred to as maintaining the *write* *barrier*. Frequent short collections may result in non-disruptive behavior, but may also increase the total CPU overhead of garbage collection. The copying of long-lived objects can be avoided by promoting them, but overzealous promotion leads to accumulation of garbage in older generations. Collection of older generations can be disruptive because they are traditionally large and contain significant amounts of reachable data. Such collections have the poor virtual-memory behavior and disruptive interactive response that generation collection was designed to avoid.

In the past two years generation garbage collection has been added to two commercial implementations of Common Lisp [33, 75]. It has also been proven effective in Smalltalk [17, 84] and ML [3]. While generation garbage collection appears effective in today's computer systems, changing systems technology will almost certainly require new garbage collection algorithms.

1.3.5 1990's – Systems of the Future

Two significant technology trends have developed recently that represent potential challenges to existing garbage collection technology. The first trend is the availability of high-performance RISC architectures and workstations based on this technology. These systems will place increased performance demands on garbage collection algorithms in the absence of hardware support. For example, where today's MC68020 processors are capable of allocating 100,000 bytes per second, new processors, such as the Intel 860, may be capable of 10–50 times that rate. While cache memory speeds will also increase, the delivery rate of large main memories is likely to remain relatively slow. In addition, the latency of magnetic disks is likely to remain as it has for that past ten years (i.e., from 10–30 milliseconds). Fast processors will place a greater demand on the memory system, and as garbage collection is intimately tied to memory system performance, garbage collection algorithms will need to adapt to best fit the technology.

Another obvious technology trend is the commercial availability of shared-memory multiprocessors. In this thesis, the term multiprocessor refers exclusively to shared-memory systems. Multiprocessors require significant changes in garbage collection technology for two reasons: first, garbage collection algorithms must be parallelized to take advantage of multiple processors, and second, multiple processors introduce new constraints in the memory system. In particular, shared communication resources (such as a shared-memory bus or disk I/O channel) introduce new potential bottlenecks to performance. Little about Lisp applications running on multiprocessors is known. Several dialects of Lisp with features for multiprocessing have appeared [36, 40, 97], but the behavior of large applications in these dialects has not been studied because the systems have not been available for serious use, and/or are incomplete.

In the past, the proposal and analysis of new garbage collection algorithms has been ad hoc. Algorithms have been proposed when problems with existing algorithms became obvious in practice. With rapid changes in technology, precise evaluation of new algorithms is of great value. This thesis describes an effective technique for evaluating garbage collection algorithms and uses this technique to compare several important collection algorithms.

1.4 Overview of the Thesis

One major tenet of this thesis is that previous evaluation studies of garbage collection algorithms have been limited in scope, especially with respect to controlled comparison of algorithms. Chapter 2 describes qualities of a good evaluation study, introduces metrics on which evaluations should be based, and describes previous work in evaluating garbage collection algorithms.

Chapter 3 introduces a trace-driven evaluation technique called *object-level tracing* that can used to evaluate the performance of garbage collection algorithms, and, more generally, the performance of runtime systems. Object-level tracing provides extensive information about all aspects of the performance of the algorithms simulated. Performance metrics of interest include CPU overhead, the distribution of disruptive pauses, and the effect of garbage collection on the reference locality of the program. Chapter 3 also introduces the test programs used in the performance studies reported in the thesis.

One advantage of object-level tracing is that it allows controlled comparisons of very different garbage collection algorithms. Chapter 4 describes the three garbage collection techniques being evaluated: stop-and-copy, incremental, and mark-and-deferred-sweep collection, all defined with extensions for generation garbage collection. In implementing each algorithm, several design decisions must be made about their configuration. Chapter 4 defines the design parameters of interest and indicates their impact on performance.

Having defined the evaluation method and the measurements of interest, I examine different performance aspects of each of the algorithms in Chapters 5, 6, and 7. Chapter 5 investigates the CPU overhead needed to implement each algorithm, considering the base cost of each approach, and the additional costs necessary to maintain invariants such as the read and write barriers. Alternative implementations are considered to determine the lowest cost approach. I conclude that incremental collection on stock hardware can be expensive, but that increasing the semispace size can considerably reduce the overhead of certain implementations. Furthermore, I find that mark-and-sweep collection with generations can be implemented with approximately the same CPU overhead as stop-and-copy collection.

Chapter 6 examines the reference locality of different algorithms both in the main memory (macro locality) and in the CPU data cache (micro locality). Using stack simulation techniques, I measure the page fault rate and miss ratio for a large number of memory and cache sizes in a single pass of the reference string. One important conclusion is that small generations can significantly improve cache miss ratios at a cost of larger execution overhead. In computers where the memory system is a bottleneck, small generation sizes may be very effective. Another surprising conclusion is that the locality of reference of incremental garbage collection is not significantly different from normal copying collection for reasonable cache and memory sizes. Finally, the results in Chapter 6 indicate that mark-and-sweep collection provides significantly better reference locality than copying collection, and requires 30–40% less total memory to achieve the same page fault rates.

The third performance aspect of garbage collection examined is the pause length, and as a related subject, the amount of data promoted during collection. Pause length is related to the number of reachable objects in the semispace and pause frequency is related to rate of allocation and the amount of time taken to fill the semispace with data. The promotion rate determines how fast older generations fill with data, and since the time needed to collect these older generations is substantial, promotion greatly impacts the interactive performance of Lisp systems. Chapter 7 presents the lifespan distribution of objects as a function of object type and application. Using this information about object lifespans, I predict the duration and frequency of garbage collection pauses for a variety of collection parameters including promoting policy and semispace size. Because incremental algorithms avoid long pauses, an important question this chapter addresses is whether or not non-incremental algorithms will provide adequate interactive performance in Lisp systems of the future.

Having looked at performance of algorithms in existing systems, Chapter 8 considers collection performance in systems ten years in the future. These systems will probably have processors 100 times faster than today's Sun4 workstations, and multiprocessors will have up to 100 processors or more. Chapter 8 examines what demands these new systems will place on garbage collection algorithms. In particular, faster systems show a greatly increased memory demand and the advantages of mark-and-sweep collection appear more important in these systems.

As mentioned, multiprocessor technology represents major challenges to existing garbage collection designs. Chapter 8 surveys Lisp language extensions for multiprocessing and proposed methods for using multiple processors for garbage collection. Currently, experimental results from multiprocessor Lisp systems are scarce. Due to the limited nature of the experimental data, conclusions about such systems are necessarily weak. In this chapter, I indicate how object-level simulation could be used when more robust multiprocessor Lisp systems are available, and discuss what conclusions can be reached based on data that is currently available.

Chapter 9 summarizes the conclusions reached in the thesis and evaluates the methodology used. It also proposes additional performance studies that would enhance the results of the thesis and discusses extensions of object-level simulation for examining other aspects of runtime system performance.

Chapter 2

Performance Evaluation of Garbage Collection

The evaluations contained in the rest of this thesis are based on a new evaluation method described in the next chapter. Performance evaluation studies are carried out with the goal of improving aspects of performance. This chapter describes the particular goals of the evaluation methods and the performance metrics of interest when evaluating garbage collection algorithms. One conclusion of this thesis is that previous evaluation studies of garbage collection have been limited in scope and have not adequately characterized the performance of widely used algorithms. This chapter describes previous evaluation efforts and indicates the limitations of these results.

2.1 Goals of Performance Evaluation

The performance evaluations in this thesis were conducted with three major goals: to make controlled comparisons so that the performance effects of isolated parameters can be determined, to allow easy exploration of the design space so that parameters of interest can be quickly evaluated, and to provide information about parts of the design space that are not easily implementable.

As with other experimental sciences, hypotheses about performance can only be tested if experimental conditions are carefully controlled. For example, to accurately compare non-incremental with incremental copying garbage collection, other algorithm parameters, such as semispace size, promotion policy, allocation policy, and copying policy must be held constant. Furthermore, the Lisp systems in which the algorithms are implemented must be identical. Comparing incremental collection on a Lisp machine to stop-and-copy collection on a RISC workstation would provide little information.

A second characteristic of an effective evaluation method is its ability to allow easy exploration of the space of design possibilities. In the case of garbage collection evaluation, new algorithms should be easy to specify, parameterize, and modify. Parameters that govern the behavior of the algorithms should be easy to introduce and change. Examples of such parameters include semispace size, physical memory page size, promotion policy, and the number of bytes in a pointer.

A good evaluation method will answer questions about systems that do not exist or are not readily implementable. If technology trends indicate certain systems are likely to be of interest, performance evaluation should help guide future system design. In the case of garbage collection, several trends have already been noted. In particular, garbage collection evaluation techniques may help guide computer architects in building effective memory system configurations. In the case of multiprocessors, evaluation methods that predict an algorithm's performance without requiring its detailed implementation on a particular multiprocessor will save much implementation effort. If a technique for evaluating garbage collection algorithms can provide these capabilities, then a much broader understanding of the performance tradeoffs inherent in each algorithm is possible.

2.2 Metrics of Garbage Collection Performance

In traditional evaluation studies, total throughput or time to completion is the most important performance measure. With garbage collection algorithms, throughput is related to two important metrics: CPU overhead and program locality of reference. Reference locality is a significant metric because poor locality can cause memory systems to thrash, greatly reducing overall performance. In addition to throughput, Lisp systems have to provide reasonable interactive response, so in evaluating garbage collection algorithms, interactive response must also be considered.

The CPU overhead of garbage collection can be broken down into three parts:

$$T_{gc} = T_{base} + T_{refs} + T_{rep}$$

T_{base} is the time required to traverse and preserve reachable objects. T_{refs} is the additional time required to maintain garbage collection invariants with each reference. For generation collection, T_{refs} includes the time to maintain a list of intergenerational pointers (the write barrier), and for incremental collection, T_{refs} includes the time to transparently transport objects from fromspace (the read barrier). T_{rep} is the additional execution time caused by a choice of object representation convenient for garbage collection. In particular, the mark-and-deferred-sweep algorithm described in this thesis represents a vector as a fixed-sized header with a pointer to a variable-sized body (similar to the KCL garbage collector [93]). This representation requires an extra level of indirection in references to the contents of a vector.

A second metric of interest is locality of reference. This thesis investigates reference locality at two levels: macro locality in the main memory and micro locality in the data cache. Because Lisp systems are traditionally memory intensive, locality of reference has always concerned garbage collection algorithm designers. In the past, main memory locality has received the most attention. This thesis shows that locality in both the cache and main memory can contribute significantly to performance.

There are several ways to characterize the macro locality of an algorithm. The page fault rate indicates the frequency of page faults for a specific memory size. The working set size measures the number of distinct pages referenced within a certain number of references. The working set characterizes what the program *needs*, the page fault rate characterizes how a program *performs* given particular constraints. This thesis uses both of these metrics to evaluate the locality of collection algorithms in the main memory.

Traditionally, macro locality has strongly influenced the design of garbage collection algorithms. Because the cost of a page fault is several orders of magnitude larger than an ordinary memory reference, significant computation to avoid page faults can be cost-effective. Close interaction between the Lisp memory management system and the operating system virtual-memory manager can reduce page fault rates, as shown for the Symbolics [61]. Fateman suggested that user hints about page replacement strategies would improve the virtual-memory performance of garbage collection in Franz Lisp [32]. Shaw has also suggested small modifications to traditional VM systems that improve the interaction of garbage collection and virtual memory [73].

But generation collection techniques have greatly improved the performance of garbage collection algorithms [61, 57, 84]. While macro locality is still important, it does not represent the performance bottleneck it was ten years ago. Since then, processor performance has increased tremendously while memory systems have become much more sophisticated in order to deliver the instructions and data to the processor fast enough.

The locality of reference in the CPU data cache is also of interest when designing garbage collection algorithms. Locality measurements at this level of granularity revolve around the transactions between the cache and the main memory. The most common metric, the cache miss ratio, is the fraction of references to the cache that require access to main memory. Each cache miss results in a bus transaction that delivers a new cache block to the cache. Other characterizations of micro locality include bus utilization (especially important in determining performance of shared-memory multiprocessors), the bus traffic ratio, the bus transfer ratio, and the transaction ratio [83]. Because the miss ratio is intuitive and translates almost directly into additional execution time (each miss requires a fixed number of cycles to service), this thesis uses it as the primary cache locality metric.

The micro locality of garbage collection algorithms has been almost entirely ignored. One reason for this lack of interest was that the cost of poor main memory locality (10–30 milliseconds per page fault) is much larger than the cost of poor cache locality (typically 1–10 microseconds per bus transaction, depending on block size and bus speed). In addition, with the small cache sizes prevalent ten years ago (e.g., the VAX 11/780 has a 512 byte mixed instruction and data cache), garbage collection algorithms could make little difference in cache performance.

But cache locality is now of much greater interest for the following reasons. First, generation garbage collection algorithms have significantly improved the macro locality of programs so that macro locality is not necessarily a bottleneck any more. Second, cache sizes have grown significantly in recent years to the point that a garbage collection algorithm can significantly affect cache performance. Finally, and most importantly, bus utilization is a severe bottleneck in the performance of shared-memory multiprocessors with a shared-bus architecture. This thesis investigates the effect of micro locality on overall performance and determines if micro locality is an important consideration in garbage collection algorithm design.

A final measure of interest is the interactive response of an algorithm. The degradation of interactive response can be characterized by the frequency and duration of pauses caused by garbage collection actions. Unlike throughput, which is a purely objective measure, what is considered acceptable interactive response is quite subjective. For example, a strong position might hold that any noticeable pauses are unacceptable. A more reasonable position relates the frequency and duration of collection pauses. If pauses are frequent (e.g., every ten minutes or less), they should be fast enough that the user does not notice them. If pauses are less frequent (e.g., every hour of so), they may be noticeable, but should last less than a second. Pauses that last a minute or more should occur very infrequently (once a week perhaps). This thesis predicts the duration and frequency of several generation collection algorithms and determines if the interactive response provided by these algorithms is acceptable by some reasonable standards.

2.3 Previous Evaluation Studies

Previous evaluation studies of garbage collection fall roughly into three categories: implementation reports, analytic studies, and simulation-based evaluation. This section reviews the work done in each category and indicates why previous work is incomplete.

2.3.1 Implementation Reports

This category is the most common form of garbage collection algorithm evaluation. A particular algorithm is implemented in the context of particular hardware and software, and a report is written to confirm that the implementation of a proposed algorithm was successful. There are many examples of such reports. This section mentions a few of the more recent studies.

Several implementation reports describe the observed performance of particular implementations of generation garbage collection. Ungar describes the performance of generation scavenging in Smalltalk [84]. His performance metrics include CPU overhead, pause length and resident set size. The performance of ephemeral garbage collection is described for the Symbolics 3600 [61], LMI Lambda [57], and TI Explorer [25] Lisp machines. In these studies, CPU overhead and paging performance is measured. Pause length is not of interest because of the incremental nature of ephemeral garbage collection. Implementation reports of garbage collection on conventional architectures have also been published. Sobalvarro describes an implementation of generation garbage collection for the MC68020 [75]. Appel, Li, and Ellis describe the performance of a hybrid incremental garbage collector for ML using stock hardware [2]. Shaw describes a generation scheme that uses easily implemented hooks in a traditional VM system [73]. These reports consider CPU overhead and VM performance.

In helping to understand garbage collection algorithms, implementation reports have both advantages and disadvantages over other approaches. The greatest advantage of measuring an actual implementation is that real performance is measured. Commonly-used programs can be executed and timed and the results are unequivocal. Any other performance evaluation technique must make a set of assumptions, and the more assumptions made, the more likely some assumption will break down in practice. If the assumptions are incorrect, conclusions based on the assumptions may also be incorrect.

Evaluation through implementation has severe disadvantages as well. First, comparative evaluation through implementation is time-consuming and almost never done. Implementing a sophisticated garbage collection algorithm in the context of a Lisp system, operating system, and hardware configuration is difficult for the same reasons that explicit management of memory is difficult (outlined in Chapter 1). In commercial systems, new algorithms are typically implemented several years after being proposed.[1] Correct comparative evaluation requires fully implementing several very different algorithms. No comparative evaluations of significantly different garbage collection algorithms have ever been done with implementation studies.

Another inherent limitation of implementation reports is the restricted range of system parameters that may be explored. Because the implementation is specific to a particular hardware and software configuration, exploration of parameters outside the scope of the particular system is impossible.

A final problem with implementation evaluation lies in the restricted forms of evaluation that are possible. With respect to measures of locality, implementations can give only a number of page faults or the time spent handling page faults. Cache performance is often totally inaccessible to the user. Analysis through simulation (discussed below) offers much more complete evaluation over a wide range of cache and memory configurations.

2.3.2 Analytic Evaluations

Another less common technique for performance evaluation of garbage collection uses analytic models of the algorithms to predict performance. Analytic evaluation is very different from evaluation through implementation. Instead of writing detailed implementations of algorithms, evaluators construct high-level mathematical models of the operations performed during garbage collection. Instead of running actual Lisp programs, evaluators make assumptions about how programs normally behave. Analytic studies are typically conducted

[1] Generation garbage collection was implemented in Franz Allegro Common Lisp and Lucid Common Lisp in 1988, five years after the idea was published by Lieberman and Hewitt.

to determine gross characteristics of algorithms such as worst-case memory usage or expected cost of collecting a memory cell. Sometimes lower bounds on CPU overhead can be be established.

Analytic results were used in the early 1970's to determine "reasonable" semispace sizes for copying collection algorithms. Hoare computes the cost of collection as a function of the store size (S) and the average memory in use (k) [45]. He shows that a particular ratio of S/k has optimal cost. Arnborg solves a similar problem in greater depth and mentions that the model was validated, but gives no data [4].

Baker's paper on incremental garbage collection considers the effect of the scanning parameter (k) on the maximum memory requirements of the program [6]. He concludes from this analysis that the maximum memory requirements of incremental collection are significantly larger than traditional copying algorithms and proposes "cdr-coding" as a mechanism to regain some of the lost address space.

Wadler examines algorithms for real-time on-the-fly garbage collection in the style proposed by Dijkstra [87]. He concludes that the worst case performance of such a system is only two times the overhead of traditional garbage collection algorithms. Hickey and Cohen investigate the performance of proposed on-the-fly algorithms by describing allocation and collection processes in terms of conditional difference equations [41]. They conclude from this analysis that mutator/collector pairs would exhibit one of three types of qualitative behavior. The values of a few system parameters such as ratio of active to total storage (similar to Hoare's parameter S/k) can be used to determine the efficiency of parallel collection.

In all of these studies, a measure of the program's locality of reference is not included as part of the model. This incompleteness is a characteristic of analytic performance studies. Because gross properties of the system are captured as parameters to the model (e.g., rate of reference, rate of allocation, etc.), effects that result from microscopic actions (like a memory reference) are impossible to predict. In practice, only one aspect of performance, either CPU overhead or memory utilization, is modeled in each study.

A second problem with analytic performance evaluation is the number of simplifying assumptions made. Typical assumptions include a constant rate of allocation, a constant rate of deallocation (i.e., a steady-state amount of memory allocated), a constant marking rate, etc. In practice, such assumptions are often incorrect. While not completely invalidating the results, these assumptions decrease one's confidence that the results are correct.

2.3.3 Evaluation through Simulation

The most promising approach to evaluating garbage collection algorithms is through trace-driven simulation. Occupying the middle ground between implementation and analytic models, simulation provides believable results without the high cost of implementation. In addition, simulation allows flexible parameterization of the execution environment to allow diverse exploration of the space of design parameters. Surprisingly, simulation has been the least common evaluation technique for garbage collection. Furthermore, only a handful of the limited simulation results are based on trace-driven simulation. Thus, simulation appears to be an effective and underutilized tool for performance evaluation of garbage collection algorithms.

Simulation of garbage collection using synthetic workloads has been used sporadically for many years. Baecker describes the evaluation of a page-based copying algorithm for an Algol-like language [5]. His synthetic workload consisted of randomly adding and deleting elements from a two-way linked list. Newman and Woodward describe a simulation of Lamport's algorithm for on-the-fly multiprocessor garbage collection [65]. Their synthetic workloads vary from sets of dense linear lists to highly-interconnected, sparsely-allocated graphs.

Davies combines an analytic approach with a simulation [26]. His study investigates the relationship between cell size and object lifetime in a non-compacting marking algorithm. Davies first describes his model of memory occupancy and then tests his model with computer simulations based on synthetic allocation of objects with familiar lifespan and size distributions (e.g., exponential, hyperexponential). His research predicts memory system requirements under these conditions.

Cohen and Nicolau use simulation to predict the effectiveness of compaction in several mark-and-sweep algorithms [22]. In their simulations, they construct *time-formulas* for each algorithm to predict the CPU overhead of the algorithm based on counts of different operations, such as additions, comparisons, and assignments. Cohen and Nicolau compare the CPU overhead of the algorithms, and they do not investigate other aspects of performance at all. Furthermore, they limit their comparisons to compacting mark-and-sweep algorithms, and do not consider copying, generation, or incremental algorithms.

Most recently, Ungar has evaluated the performance of generation scavenging strategies based on trace-driven simulation [85]. In his study, Ungar collects lifespan information about objects allocated over several four hour interactive sessions and writes this data to a file. He then uses the trace file to predict the effectiveness of different policies for promoting data in the context of generation scavenging. The CPU overhead and pause length are estimated. No estimate of the locality of reference is attempted.

Evaluation based on trace-driven simulation has great potential. Comparative evaluation is possible, system parameters can be varied easily, reference data is easy to collect and evaluate, and evaluation can be based on the execution of any interesting program so that few simplifying assumptions need to be made. Results based on trace-driven simulation are currently sparse. Until very recently, no one has used trace-driven simulation to study the reference locality of garbage collection algorithms. The reason for the lack of research in this area lies in the computational cost of trace-driven performance evaluation.

Evaluation based on reference traces requires that actions be taken at every memory reference event. In the execution of programs, tens of millions of such references take place. Until recently the computational cost of performing an analysis of 10^8 events has been high. Recent breakthroughs in processor and memory technology, which have provided high-performance, low-cost workstations, have significantly reduced the cost of simulation. The next chapter describes a trace-driven simulation technique that allows comparative performance evaluation of garbage collection algorithms.

Chapter 3

A New GC Evaluation Method

This chapter describes an effective technique for comparing the relative performance of garbage collection algorithms over a wide range of parameter values. The technique is trace-driven simulation at the program object level using a tool called MARS (Memory Allocation and Reference Simulator). This chapter outlines the design of MARS and describes the test programs that are used in this thesis to compare the performance of several garbage collection algorithms.

3.1 Overview of MARS

Trace-driven simulation based on program address references has been used for many years to investigate hardware and software support for memory systems. Such studies collect program address references and feed these traces through a simulator to predict memory system performance. While raw address traces are sufficient to investigate the performance of cache configurations and page-replacement algorithms, trace-driven simulation of garbage collection must be carried out at a higher level because garbage collection manipulates program objects and not raw addresses. MARS traces events as they occur to program objects, and not memory addresses. In particular, object allocation, deallocation, reference, and assignment events drive the garbage collection simulation.

MARS is a tool that attaches to an existing Lisp implementation (called the "host" Lisp system). Figure 3.1 illustrates its components. MARS avoids interfering with the execution of the host Lisp system, except that it considerably slows program execution and increases the size of the virtual memory image. Because the host Lisp system is not disturbed, any program that will execute in the host system can be used as a test program for MARS. MARS requires small modifications to the host Lisp implementation so that when interesting events occur (e.g., car, aref, or cons), information is passed to the simulator. This *implementation interface* is intended to be quite small (in Franz Allegro Common Lisp the total interface is about 300 lines of Lisp and C source code).

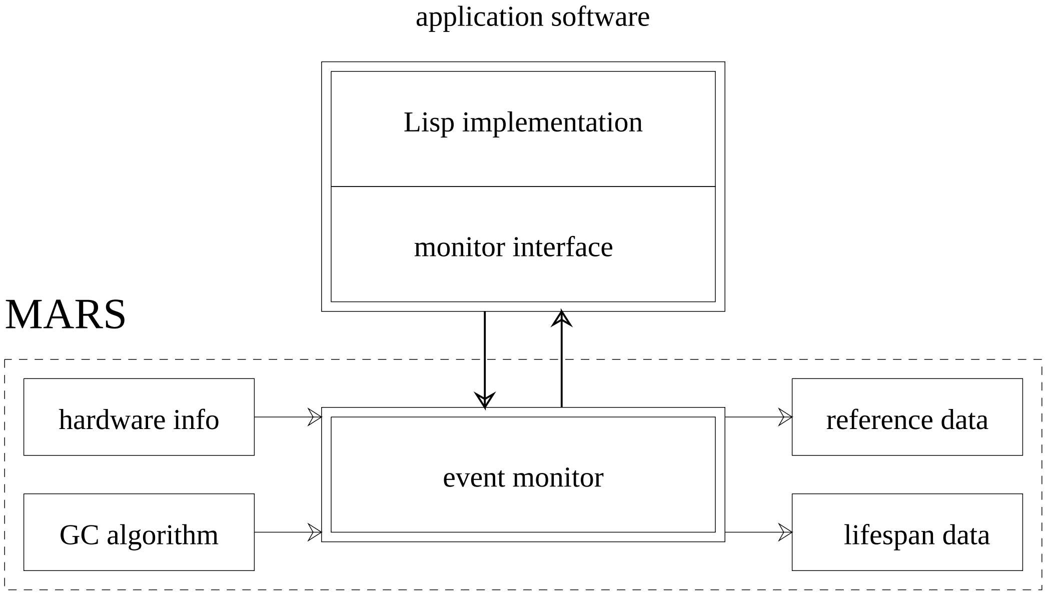

Figure 3.1: MARS Organization.

Unlike many trace-driven simulators, which store the reference events being traced in a file, MARS simulates and evaluates garbage collection algorithms on-the-fly. There are several reasons to avoid creating large trace files for garbage collection evaluation. First, there are tens of millions of reference events in a typical Lisp program execution. Storing a trace of all such events would require creating a large trace file (hundreds of megabytes) for each program of interest. Second, connectivity information (i.e., what objects point to what other objects) available in the executing program is needed by MARS to accurately simulate the behavior of garbage collection algorithms. If only reference data were written to a file, connectivity information would be lost. Finally, trace files often are needed to record the behavior of a difficult to reproduce workload (like a multi-user operating system). With MARS, programs are used to generate the traces, and so the traces are very reproducible.

3.2 Shadow Memory Simulation

MARS simulates garbage collection without any interference with the host Lisp system by maintaining its own view of how program objects are allocated in memory. This simulator view of memory is called the *shadow memory* because operations on objects in the host Lisp system are "shadowed" in the simulator's view of memory (Figure 3.2). By separating the host implementation's view and the simulator's view of where objects are allocated in memory, the simulator is free to allocate and transport objects in the shadow memory independent of the host Lisp system. In reality, the shadow memory is not actually implemented

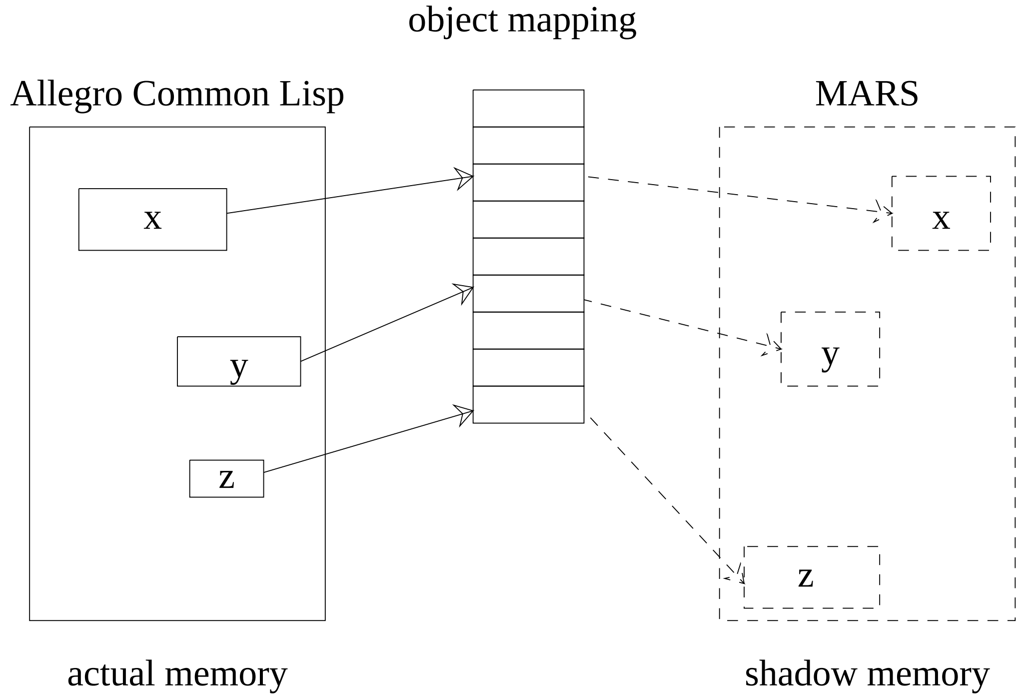

Figure 3.2: Shadow Memory Mapping.

except as a mapping from host Lisp objects to shadow memory addresses and vice versa. In MARS, the shadow memory is implemented as a pair of large hash tables pointing to a collection of structures containing shadow memory locations, object size and type, object time of birth, and a count of the read and write references to the object. Because a Common Lisp implementation will normally have 200,000 or more objects active at any time, maintaining the shadow memory and related data can be quite memory intensive. All eight test programs described in this thesis had virtual images larger than 40 megabytes.

Shadow memory simulation is also CPU intensive. Every Lisp program object reference (e.g., car, symbol-value, aref) needs to be translated to a reference in the shadow memory for correct simulation. Thus, an operation that would normally require a single memory reference now requires several function calls and a hash table lookup. On top of that, if stack simulation is being used to measure locality of reference, each reference can be slowed even further.

3.3 Measuring Time with References

One problem that arises in many experiments is interference between the measuring apparatus and the events being measured. Even at larger than atomic scales, an "uncertainty principle" takes effect. Clearly if shadow memory simulation slows execution time by a factor of twenty or more, program CPU seconds cannot be used to measure time. Reported CPU time has other inherent limitations. Often the resolution of processor time available from the operating system is not very fine (15 milliseconds) and the durations of interest (such as an object lifespan) can be much smaller. As an alternative, Shaw measures object lifespans by noting how many garbage collections an object survives [73]. Assuming a constant rate of allocation, this technique can only provide coarse-grained measurements since at most few garbage collections happen each second. Memory allocation, however, is highly dependent on the program being executed, so that the assumption of a constant rate of allocation may be invalid. Later is this chapter, I show that the rate of allocation in my test programs varies widely in a non-random fashion.

As an alternative, a count of heap memory references provide a good standard unit of time and have been used as a unit of time in many performance evaluation studies. Using individual references as a unit of measure allows measurements with microsecond resolution, and the assumption that references occur at a constant rate is experimentally sound. Figure 3.3 shows the rate of heap references per instruction for three large programs and several small benchmarks. While the rate of reference is not a constant, the variance about the mean is quite small compared to the variance in the rate of allocation. The small benchmarks, taken from the Gabriel benchmark suite [34], indicate how these benchmark programs show almost no variation in rate. The small benchmarks show little variance because they perform the same task repeatedly, which is not how larger programs normally execute. Even though the larger test programs show more variance, the variation is sufficiently small that an assumption of a constant reference rate is reasonable. Furthermore,

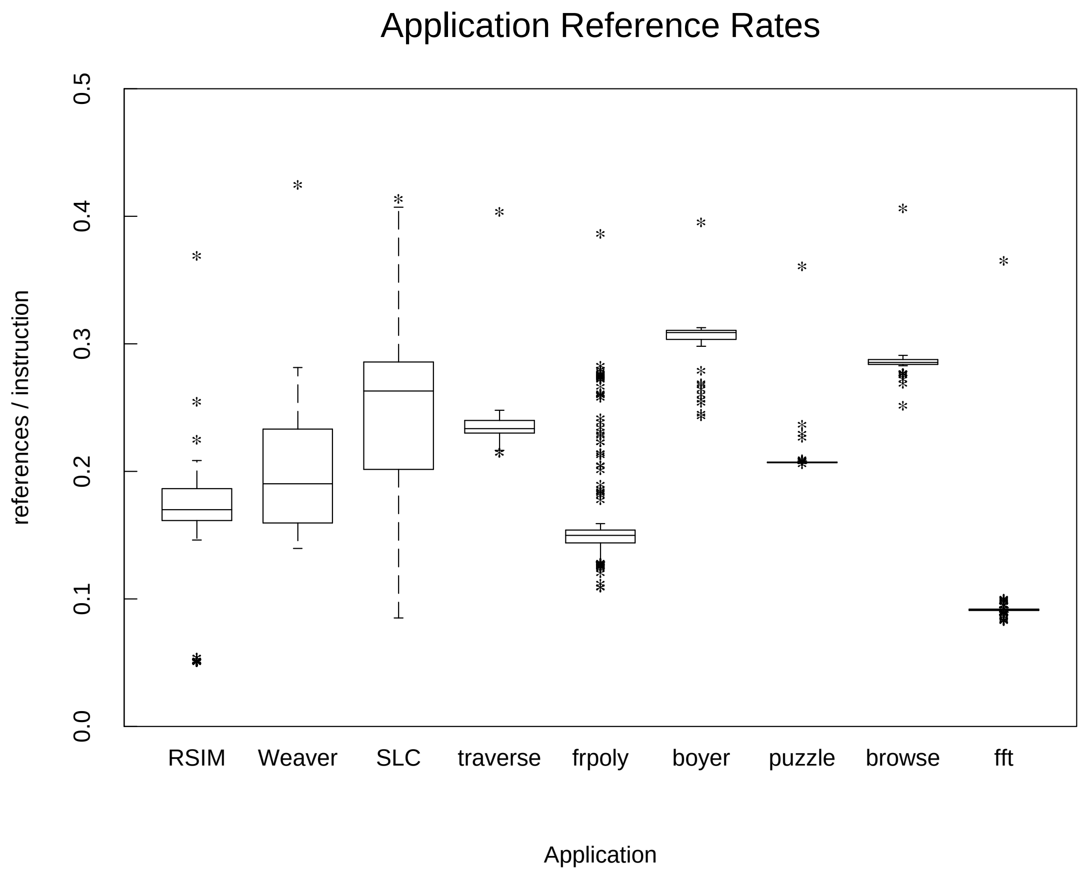

Figure 3.3: Reference Rate Distributions for Several Programs. The distributions are presented using box plots. The box indicates the 25th, 50th (center line), and 75th percentiles. The lines extending above and below each box indicate values in a 1.5 interquartile range. Points outside this range are considered outliers and marked individually.

the variation indicates that the programs are exhibiting a variety of behavior in their execution. The rest of this thesis assumes that programs show a constant rate of reference to the heap and durations are calculated based on the assumption that a count of memory references can be used as a clock.

3.4 Reference Locality Based on Heap References

The single event type that MARS uses to predict all other performance metrics is the heap memory reference. Lisp programs also reference memory to load instructions and access the stack. Table 3.1 breaks down memory accesses by type for four SPUR Lisp applications. The relative frequency of instruction fetchs, heap reads and writes, and stack reads and writes is shown.

Memory Access Type	RSIM	Weaver	SLC	PMA
Instruction fetch (%)	86.5	87.5	83.1	83.2
Heap reads (%)	7.7	10.6	12.2	10.6
Heap writes (%)	0.8	0.6	1.4	1.3
Stack reads (%)	2.1	0.6	1.7	2.4
Stack writes (%)	2.4	0.6	1.7	2.5
Heap accesses / instruction (%)	9.8	12.8	16.4	14.3
Ifetches / Heap Reference	10.2	7.8	6.1	7.0

Table 3.1: Relative Frequency of Memory Reference Operations for four SPUR Lisp Applications. RSIM, Weaver, and PMA are described in Appendix A. SLC is the SPUR Lisp compiler.

MARS simulation does not track instruction fetches because, while they account for a lion's share of the memory traffic in a program (> 80%), their reference locality is very different from the reference locality of heap references. Furthermore, garbage collection techniques have little effect on the reference locality of instruction fetches. Many systems take advantage of this difference by providing separate caches for instructions and data (e.g., the MIPS R3000 architecture [50]). The micro locality studies presented later in the thesis assume a processor with separate instruction and data caches.

Accesses to the runtime stack are not of interest for similar reasons. It is accessed differently from the heap, and garbage collection strategies have little or no effect on accesses to the runtime stack. Furthermore, commercial systems have been designed to minimize stack-related memory references through efficient use of registers to store stack values [49, 42, 79, 88]. Table 3.1 shows that the SPUR register windows are effective at minimizing runtime stack references. In general, the effectiveness of minimizing stack-related memory traffic depends entirely on the architecture and compiler. In my simulations, I assume that runtime stack references have been entirely eliminated. In any event, stack-related memory references have much better locality of reference than heap references, so my reference locality results would be uniformly improved by including stack references. By leaving out such references, I simply provide a slightly more conservative estimate of the locality.

Shadow-memory simulation does not preclude also measuring stack and instruction memory references, but because garbage collection will not affect the locality or occurrence of these references, they were not deemed important enough to measure.

3.5 Other Tools

In addition to the heap reference-level results collected with MARS, another simulator was used to collect instruction-level results. BARB is an instruction-level simulator for the SPUR architecture [42] that was used to develop and to debug SPUR Common Lisp and to predict the performance of the SPUR architecture [81]. Because BARB has relatively good performance (50,000 simulated SPUR instructions per second on a VAX 8800), it was used to gather some of the instruction-level measurements provided in this thesis. More details about BARB are available with a description of the SPUR Lisp implementation [96].

3.6 Test Programs

The results in this thesis were collected by attaching MARS to Allegro Common Lisp executing on a Sun4/280 computer with 32 megabytes of memory. Modifications were made to the Allegro Common Lisp compiler and runtime system to intercept the necessary events (e.g., car, cdr, cons, aref). The test programs were compiled with low optimization and high safety settings to force certain operations to be performed out-of-line. Except where the compiler might have optimized away temporary numerical objects (such as intermediate floating point values), the speed and safety settings of the test programs should have no effect on the gathered data.

The programs used as test input were gathered from a variety of sources. The main goal in collecting these programs was to find large programs that represent a variety of programming styles, programming paradigms, and application areas. Computational programs were favored over interactive programs because they allocate more intensively; and with the goal of understanding the performance of garbage collection, I was most interested in stressing the performance of the garbage collector with compute-intensive programs. Furthermore, with the goal of understanding the performance characteristics of systems of the future, I am interested in compute intensive programs because they are the programs that need the performance provided by multiprocessors and fast workstations.

My eight test programs represent a range of programming styles from Maclisp programs converted into Common Lisp, to recently written Common Lisp programs that use hash tables, sequence operations, and user-defined structures. The programming paradigms include rule-based expert systems, object-oriented programming using CLOS (Common Lisp Object

System), data-driven pattern-matching, and also normal procedural programs. Applications include a theorem prover, a compiler, a circuit simulator, and a program transformation system.

Results from eight test programs are inherently difficult to display. Graphs with eight sets of lines are confusing, eight separate graphs do not fit on a page, and tables with eight columns are too wide. To solve this problem, in the body of the thesis, results from only four of the test programs are presented. A description of the four additional programs and related results are available in Appendix C. Furthermore, since graphical data is easier to understand than tabular data, the body of the thesis presents results graphically when possible. A tabular form of the results for all eight test programs is available in Appendix D.

The four programs used in the thesis body are the largest and most diverse of the eight test programs. Information about these programs is summarized in Table 3.2.

Resource Description	ACLC	Curare	BMTP	RL
Source lines	46500	45000	21500	10200
Execution time (sec), w/o monitor	410	242	211	477
Execution time (sec), w. monitor	6591	4708	6644	9202
Monitor factor slowdown	16	19	31	19
Program references (millions)	83.7	57.9	69.3	108.1
Objects allocated (millions)	5.1	1.43	1.3	7.8
Bytes allocated (millions)	59.9	16.9	11.1	81.8

Table 3.2: General Information about the Test Programs.

3.6.1 The Allegro Common Lisp Compiler (ACLC)

The Allegro Common Lisp compiler is the largest test program, containing approximately 47,000 lines of Common Lisp source code. The compiler is a widely used commercial compiler and is written in a "modern" style, using all the functions available in Common Lisp. The compiler reads the source input, generates an abstract syntax tree, generates "quads" from the tree, and then optimizes the quads. The test input for the compiler is three large files containing 8,300 lines of source code. The three large programs cause the compiler to allocate about 60 megabytes of data.

3.6.2 Curare – A Scheme Transformation System

Curare is a source-to-source program transformation system that parallelizes Scheme code for execution on a multiprocessor [54]. It is written in an object-oriented style and uses the Common Lisp Object System (CLOS) [12]. Curare reads a Scheme program, computes dependence information through both intra- and inter-procedural dataflow analysis, computes the program alias graph, which embodies structural conflict information, and applies a series of transformations that eliminate conflicts and parallelize the code. While the input program is small (less than 100 lines), Curare allocates 17 megabytes of data while transforming it.

3.6.3 The Boyer-Moore Theorem Prover (BMTP)

The Boyer-Moore theorem prover is a large Lisp program originally written in Interlisp, converted to Maclisp, and finally converted to Common Lisp [14]. It should not be confused with the Boyer benchmark program fragment reported in Gabriel [35] and widely used in estimating the performance of Lisp systems. The Boyer benchmark is a small fragment (150 lines of source) of the complete Boyer-Moore theorem prover (21,500 lines of source). The theorem prover builds a database of patterns and proves theorems by matching the theorem to be proven against known patterns. Because this program was written in an older dialect of Lisp, lists are used almost exclusively throughout the program. This program is of interest because it exemplifies two things—an application containing an AI search algorithm, and a large program written in an older style that can be contrasted with modern Common Lisp programming styles. The test input to the theorem prover is part of a proof of the Church-Rosser theorem. The proof requires significant computation, but allocates less memory than the other test programs (11 megabytes).

3.6.4 A DSP Microcode Compiler (RL)

RL is a microcode compiler for a class of digital signal processing architectures with horizontal microcode [70]. It is written with modern Lisp programming techniques and data structures. RL is unlike other compilers in that its main job is to schedule different functional units horizontally in the microcode being generated. It does this using a network flow algorithm on a graph representing function units and time. RL is also interesting because it makes heavy use of structures to represent the graph it is analyzing. Its network flow algorithm is representative of applications that manipulate large graphs. The computation performed is very memory intensive, allocating almost 82 megabytes of data while compiling two input files with a total of approximately 100 lines of text.

3.6.5 Programs in Appendix A

Data for four more programs is presented in Appendix C along with a description of each program. I briefly introduce these programs here so the reader is aware of their presence. The programs are: RSIM, Weaver, a Prolog compiler, and the PERQ microcode assembler (PMA). RSIM is a transistor-level circuit simulator simulating a simple counter [82]. It was selected because it has been used in other empirical studies [74] and because, of the test programs, it makes the heaviest use of floating point numbers. Weaver is a routing program written as a set of OPS5 rules. It exemplifies another common AI programming paradigm: rule-based programming. The Prolog compiler compiles a Warren abstract machine (WAM) representation of Prolog into instructions for a RISC architecture (SPUR). This compiler performs several optimization phases by pattern matching components of a tree representing the program. PMA is the PERQ microcode assembler for the Spice Lisp system [89].

3.7 Characterizing the Test Programs

This section presents general data gathered using MARS that characterizes the behavior of the test programs. By first understanding their general behavior, we can then better understand how garbage collection will affect this behavior.

3.7.1 Object Allocation by Type

Previous studies have measured large Lisp programs and found that cons cells were the most frequently allocated object [19, 74, 78]. Shaw expressed concern that the programs he measured did not represent the most up-to-date Common Lisp programming practices, including use of the wide variety of data types available in Common Lisp. Most of the programs were written by programmers who were fully aware of the data types available in Common Lisp. Most contain user-defined structures. Of the four programs presented, only the Boyer-Moore theorem prover was not originally written in Common Lisp.

Figures 3.4 and 3.5 present allocation information for each test program broken down by type. Figure 3.4 show the fraction of total bytes allocated for each class of types. Figure 3.5 shows the fraction of objects allocated for each class. These figures (and subsequent ones) break down types into five categories: cons cells; symbols; vectors, which include arrays, vectors of general objects, and user defined structures; numbers, which include floating point numbers, bignums, complex numbers and ratios; and other, which includes all other data types, but most significantly strings and function objects.

Cons cells represent more than 50% of total space allocated, even in programs written in a “modern” style. Vectors and structures account for the second largest share, ranging from 15–30% of total bytes allocated (excluding the theorem prover). Cons cells account for more than 80% of the objects allocated in all cases. This data indicates a garbage collection algorithm should optimize the reclamation of cons cells and a storage allocation algorithm should optimize the allocation of cons cells.

This data also points out similarities and differences between the test programs. None of the programs use significant amounts of numerical data. Symbols and strings are infrequently allocated. The Boyer-Moore theorem prover is clearly different from the other programs in its lack of types other than cons. Curare, while written in an object-oriented programming style, makes relatively infrequent use of instance objects (represented as vectors), probably because of the high overhead associated with manipulating instance objects in CLOS as compared with operations on regular Lisp data types. A large part of the “other”

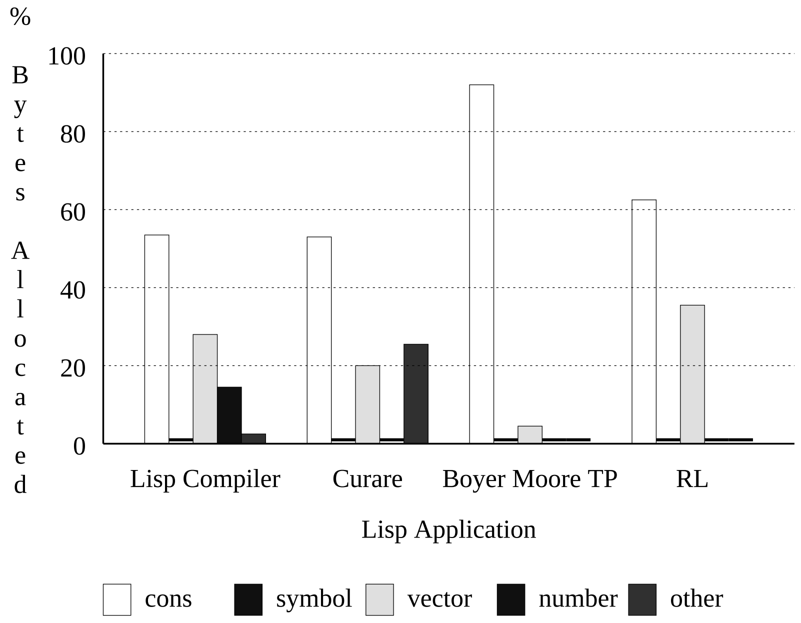

Figure 3.4: Object Allocations for Test Programs (by type and size).

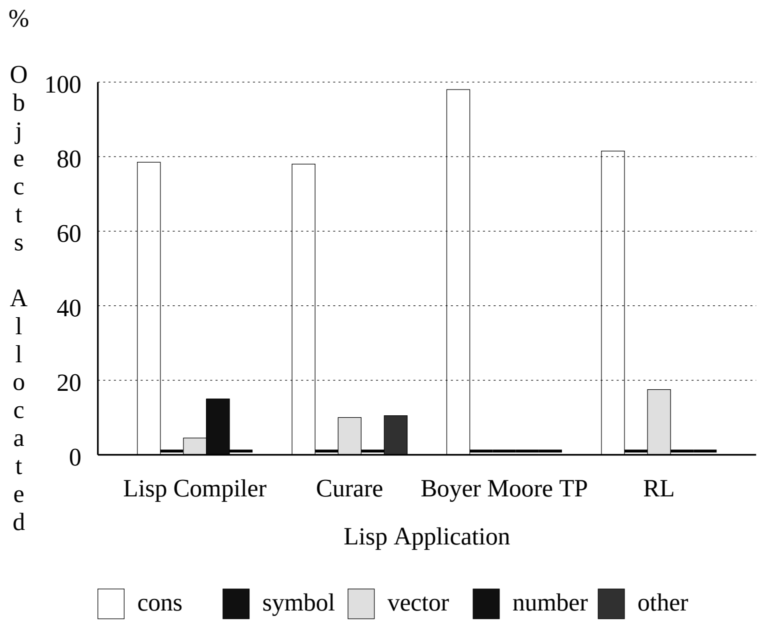

Figure 3.5: Object Allocations for Test Programs (by type and number).

component in Curare is due to dynamically defined code objects allocated by CLOS. The Lisp compiler and RL both show a healthy use of Common Lisp structures, although vectors still represent a small fraction of the total bytes allocated.

3.7.2 Object References by Type

Figure 3.6 breaks down references to objects by object type for the four test programs.

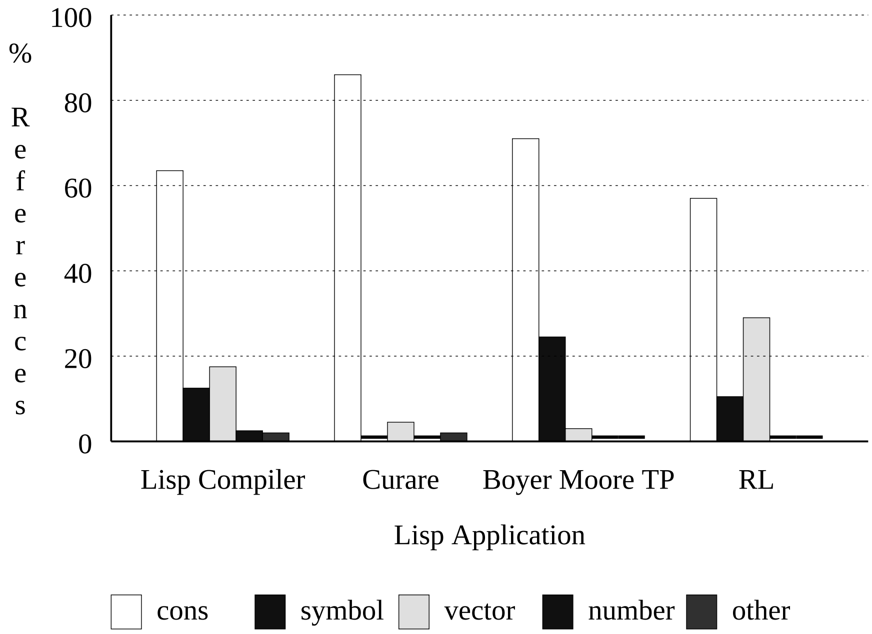

Figure 3.6: Object References by Type for Test Programs.

Again, cons cells represent the large majority of object references, although many references to symbols are also noted. Symbol references represent accesses to the symbol value of the symbols, either retrieving the value of a global variable, or saving and restoring the value during binding and unbinding of special variables. Many Lisp systems also access the function cell of a symbol at each function call to implement function call indirection. These accesses are not included in this data because they represent an artifact of the implementation. In another paper, I describe the effectiveness of direct function calls in SPUR Lisp [94].

Because cons cells remain the most frequently allocated and referenced data type, the effectiveness of a garbage collection strategy depends heavily on how that algorithm interacts with the observed lifespan behavior of cons cells. Chapter 7 presents a more detailed look at the lifespan distribution of objects. For now, I will help characterize the behavior of the test programs by looking at the rate of birth and death of object types throughout the execution of the program.

3.7.3 Object Birth and Death Rates

Rates of object allocation and deallocation are extremely application-dependent. While looking at the object birth rates for four applications does not allow us to make general conclusions about such rates, it does provide us with a better understanding of the programs being measured. Figure 3.7 presents the allocation rates of objects by type for the four test programs. Execution times presented do not correspond with measured execution, but to the normalized time as measured using program references.

Looking at the changes in allocation rates, the behavior of the programs becomes more apparent. The Lisp compiler was measured compiling three files. The birth rates show a decrease in cons allocation and an increase in number allocation when the compiler finishes compiling each file. Numbers are allocated when the FASL file is written because the 32-bit instructions are represented as bignums (arbitrary-sized integers). RL was used to compile two files of different sizes. The allocation rates clearly indicate two similar execution sequences. Curare allocates a large dependence graph in its early stages, and then allocates very little for most of its execution. Curare's author carefully tuned it to remove unnecessary memory allocation. In all cases, the birth rates indicate that the programs showed a complex time-dependent allocation behavior, where the rate of allocation varied dramatically over the execution of the program. In contrast, smaller benchmarks, such as those in the Gabriel benchmark suite [35], show a near-constant rate of allocation as a function of time because they represent small programming tasks repeated hundreds of times. Such simply-behaved programs are not suitable for effective trace-driven simulation studies.

Another conclusion we can draw from these graphs is that while average allocation rates on the order of 500 kilobytes per second should be expected, peak rates several times higher can be sustained for periods of several seconds or more.

In addition to allocation rate, deallocation rate is also of interest to designers of storage reclamation systems. Figure 3.8 shows the *net* allocation rates for the four test programs as a function of time. The net allocation rate is measured as the difference between the number of bytes allocated and bytes deallocated in a particular time interval. Objects are assumed to be deallocatable immediately after the last reference to them occurs before they are reclaimed.

Many analytic models of garbage collection assume allocation and deallocation are in a steady state (i.e., equal), which means the net allocation rate is zero. The figure shows that such an assumption is false if the time interval considered is sufficiently small (on the order of a few seconds). The figure indicates that periods of rapid heap growth are often followed by phases of heap contraction. Furthermore, the contractions tend to be more extreme than the expansions. Intuition suggests that extreme contractions will occur at stages in the program where a task has been completed and the memory allocated for it is not longer needed. These results enhance this intuition in two ways. Contractions do occur, but are not as extreme as one might guess (see the Prolog compiler in Appendix C for the most extreme case). Also, before contractions, periods of intense expansion often occur.

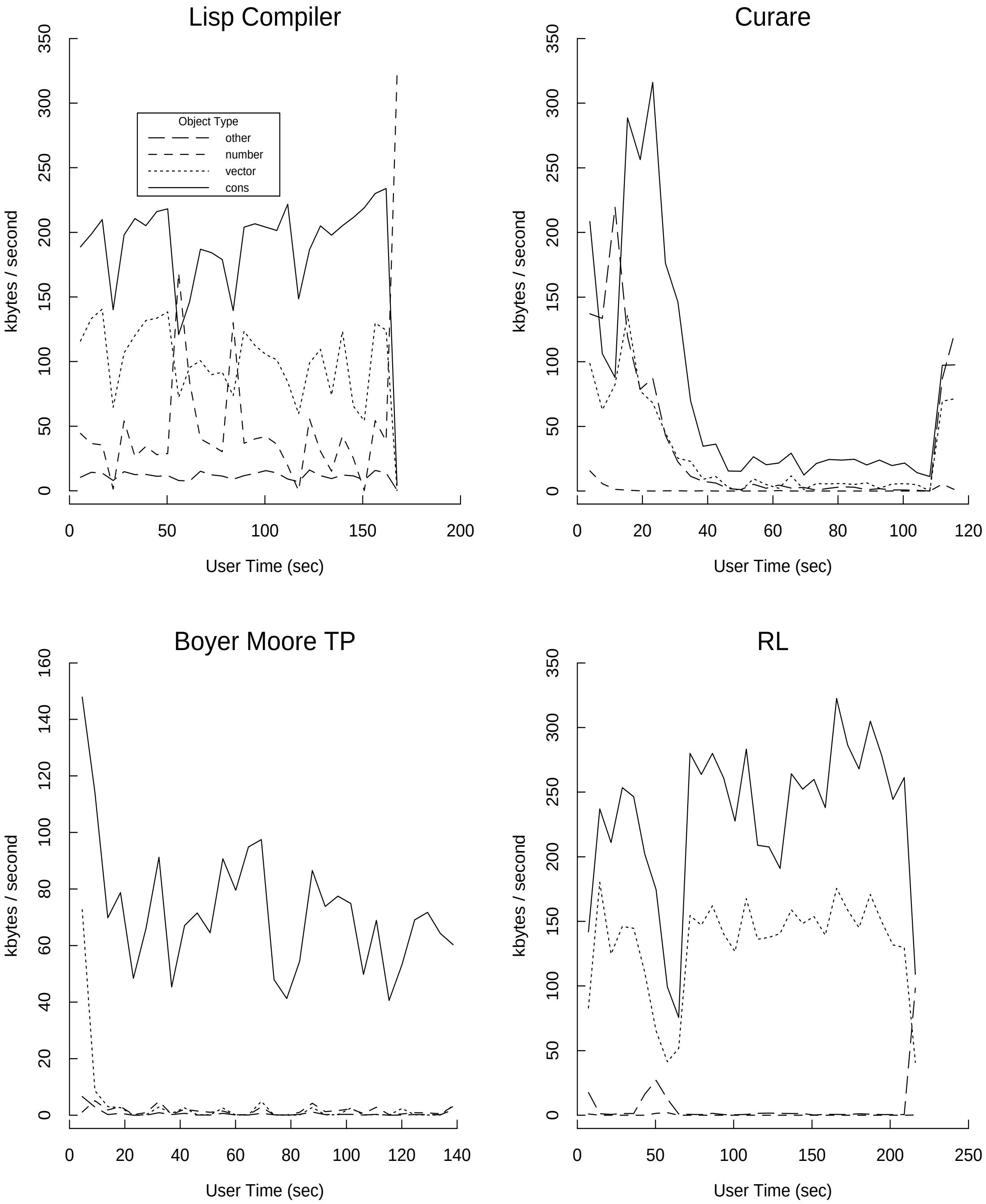

Figure 3.7: Program Allocation Rates as a Function of Time.

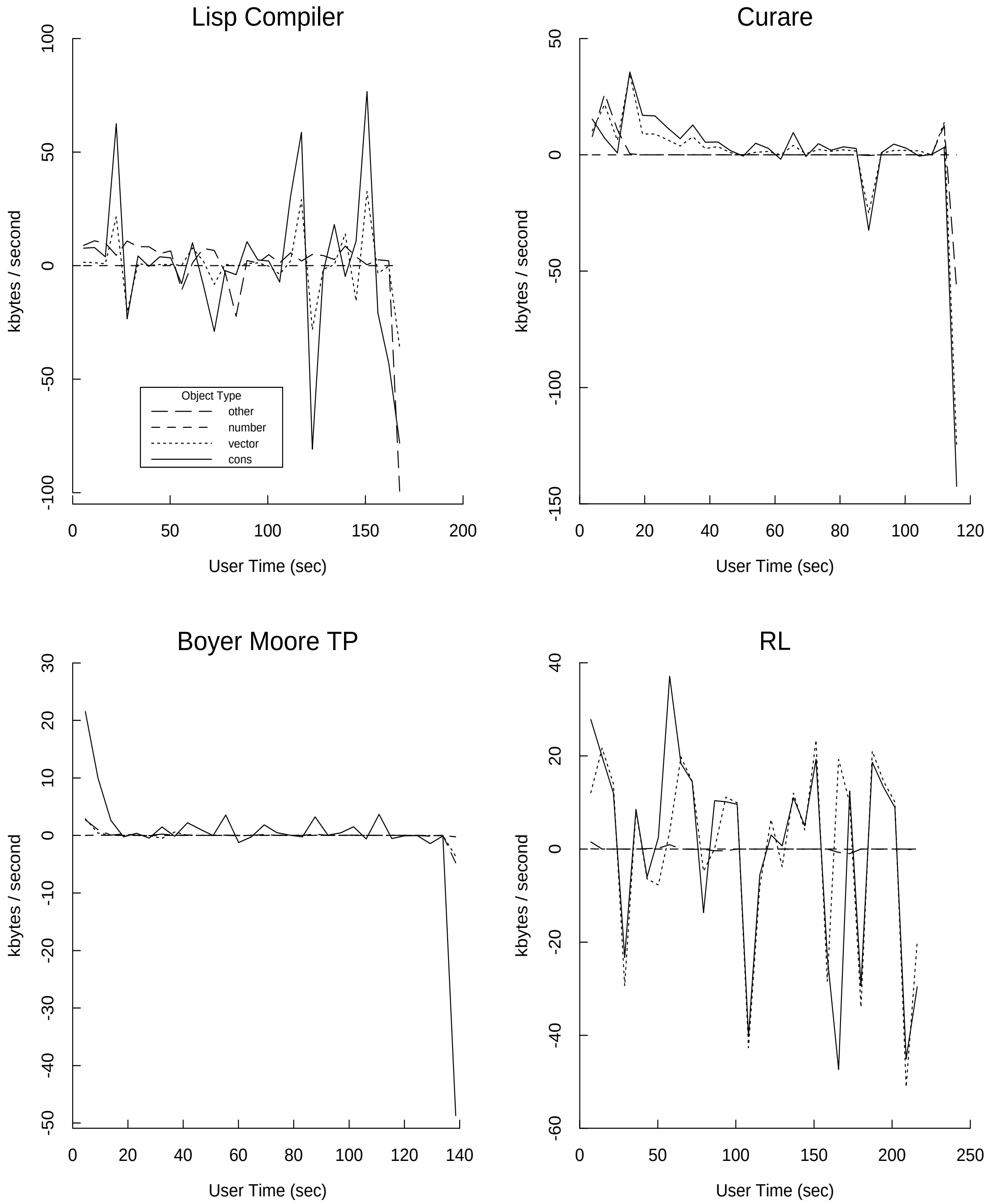

Figure 3.8: Net Allocation Rates as a Function of Time.

Garbage collection during an expansion is less effective because much of what has been allocated remains reachable. Collecting after a contraction is more cost-effective because more storage is reclaimed. Using a technique called "opportunistic garbage collection," Wilson suggests basing collection on factors such as stack depth [90]. If a low stack depth corresponds with periods of contraction, such a mechanism could be effective in enhancing the performance of collection.

Chapter 4

Algorithms for Garbage Collection

This chapter introduces the policies and parameters of interest when evaluating garbage collection algorithms and then describes the three algorithms used for the evaluation studies in the following chapters. Collection policies involve decisions about traversal, preservation, promotion, allocation, and garbage collection invocation. Three algorithms described later in this chapter that use these policies are: simple stop-and-copy collection, incremental collection, and mark-and-deferred-sweep collection. Each algorithm is augmented with generations.

4.1 Garbage Collection Policies

Policies for garbage collection can be split roughly into two groups: allocation policies, including how the heap is organized and when reclamation is invoked, and reclamation policies, including how to traverse reachable objects and how to preserve them once they are discovered. Deciding when to promote an object is an important policy for generation-based collection algorithms. This section discusses each policy in detail and suggests why some policies are of particular interest.

4.1.1 Heap Organization

Heap organization has many different facets, all of which affect the performance of garbage collection. The section discusses three aspects of heap organization: impact of data types on organization, organization of individual generations, and the relationship between generations.

Data types strongly influence heap organization. To obtain various performance benefits, objects of the same type have been allocated together in many systems [31, 13, 21, 93]. Interlisp allocated pages of cons cells and kept per-page free lists [13]. Recognizing that the value cell of a symbol is accessed more often than the other components, Cohen suggested separating the different components of symbols and putting them in different parts of memory [21]. Shaw reaffirmed the advantages of this representation [74].

Objects are often segregated by type to facilitate type identification. In systems where the upper bits of a pointer are used as a type descriptor (high tagging), as in Spice Lisp [89], objects of different types exist in different parts of the address space. Pages of objects are sometimes allocated together and a single descriptor for the page types all the objects in it, as in Franz Lisp [31]. This so-called Big-Bag-Of-Pages (BiBOP) method allows type descriptors to be maintained without using address bits. The disadvantage of storing the descriptor in memory instead of with the pointer is that a memory reference is required to identify the type of any pointer. An alternative tagging mechanism stores the tag for the most common data types (cons cells, symbols, fixnums, etc.) in the low bits of a pointer (low tagging). Recent commercial Lisp systems use low tagging to implement type descriptors (e.g., Franz Allegro Common Lisp [33]). Low tagging does not require a memory reference for type determination of most pointers, and does not reduce the size of the address space by using up the upper bits of pointers. Furthermore, low tagging eliminates the need to group objects by type. Because low tagging appears to be the most effective tagging mechanism for stock hardware, this thesis assumes low tagging is used to type pointers in the algorithms simulated.

Objects are also segregated by type to make garbage collection easier or more effective. Non-compacting mark-and-sweep garbage collection becomes complicated when data types with variable size (e.g., vectors) are allowed. If all object types are allocated together, room must be found for objects of arbitrary size. The fragmentation that results from best-fit or first-fit allocation policies may be unacceptable. If fixed-size objects of the same type (such as cons cells) are allocated in groups, then fragmentation for objects of that type can be eliminated. To minimize fragmentation even further, vector objects can be allocated in two parts: a fixed-size header that points to a variable-sized body allocated in another part of the heap. KCL organizes vectors in this way [93]. The header is reclaimed with a mark-and-sweep technique and garbage in the region of the heap containing the body is reclaimed using copying collection. While this method avoids fragmentation, it has the disadvantage that references to vector bodies must be made indirectly through the header. Because non-compacting mark-and-sweep algorithms seldom transport objects (except vector bodies), they exhibit potentially higher locality of reference. One of the algorithms considered here, mark-and-deferred-sweep collection, uses such an allocation policy.

The heap may also be organized to facilitate the scanning phase of a copying collector. The collector has to linearly scan memory and traverse pointers it finds during the scan. Some architectures, such as the Symbolics [62], maintain a bit per word of memory indicating whether the word contains boxed data (a pointer) or unboxed data (non-pointers, e.g., an instruction or floating point number). On stock hardware, the collector must maintain enough information about objects to make this decision. A standard header that indicates the size of the object can be placed with each object, so that successive objects can be identified. This method wastes memory if the header is required in small objects such as cons cells (typically 8 bytes). Often, objects are segregated by type to facilitate scanning.

Because cons cells are so common, separating cons cells from other objects and attaching a header to all types except cons cells is an effective way to facilitate memory scanning.

Increased algorithm complexity is a concern that balances the benefits of segregating types in the heap. In many cases, type specific organization represents a tuning of algorithms to improve performance. A simpler approach allocates objects of all types together in the heap (a mixed heap). For copying collection algorithms, because there are so many policies to consider, I limit investigations to consider a mixed heap, the simplest organization policy. The simulated mark-and-deferred-sweep algorithm groups objects by type to avoid fragmentation, as described above. Further studies are needed to determine if a more complex type-based organization of the heap significantly improves copying collection.

Another important consideration in heap organization is how to organize the generations and semispaces in the heap. With generation collection, several important questions immediately arise. One important parameter is the number of generations used. While Ungar maintains only two are generally necessary and more can be difficult to tune [85], Caudill reports on a Smalltalk system with seven generations and concludes that seven generations appear sufficient, although probably not necessary [17]. The SPUR processor provides hardware support for four generations and SPUR Lisp uses only three of these [96]. In my simulations, four generations are sufficient because not enough data is allocated by the test programs to require any more than three.

The way in which the operating system supports large, sparse address spaces affects the layout of generations in the heap. Many older operating systems do not support arbitrarily large holes in the address space (e.g., Unix 4.2 BSD and Ultrix). With this constraint, several large unfilled generations may be impractical. However, modern operating systems, such as Sprite [63] and Mach [69], support sparsely-populated address spaces, where widely separated areas of the heap can be mapped and referenced efficiently. For the purposes of these studies, I assume each generation resides in a separate part of the address space, generations are sufficiently far apart that they do not overlap, and each generation can grow independently of the others.

A common implementation of generation garbage collection divides generations into semispaces. To avoid premature promotion, an object is first copied back and forth between semispaces within a generation before being promoted. After a certain number of copies, the object is known to have existed long enough and is promoted. To correctly implement this policy, called *copy count promotion*, a copy count must be maintained with each object. Smalltalk implementations tend to use per-object copy counts because objects are often large anyway (Ungar reports the average size of a Smalltalk object is 50 bytes [84]). In Lisp, the most common objects, cons cells, are small and a per-object copy count is less practical. There are several alternatives to per-object copy counts. A crude method promotes all objects in a generation after a particular number of copies, and is called *en masse* promotion. The disadvantage of en masse promotion is that young objects are promoted with older ones. Other approaches group objects with similar ages together. Shaw suggests maintaining multiple semispace regions per generation and advancing objects from region to region as they are copied [74]. In this "bucket brigade" method, objects in the oldest region are promoted to the next generation. Wilson suggests that only a couple of such regions are necessary [91]. These approaches complicate the structure of the heap and the implementation significantly. To determine if copy counts are necessary, this thesis compares the relative effectiveness of copy count and en masse promotion strategies.

4.1.2 Deciding When to Collect

Traditionally, garbage collection is invoked whenever it is needed. In a two semispace configuration, when a semispace fills, a collection is invoked. If copied objects fill 90% of the "to" semispace after collection, then another collection will be invoked after allocating from the remaining 10% of the semispace. If 90% of tospace is filled after each of many collections, then frequent garbage collections will consume a large fraction of the CPU. This thrashing behavior resulting from rigid semispace boundaries is a source of instability in collection algorithms with fixed semispace sizes. Fixed semispace sizes may result from operating system constraints that prevent the heap from growing. With the belief that modern operating systems will allow expansion of arbitrary regions of the heap, I assume each semispace in each generation can grow arbitrarily if the collector decides to let it. Thus, decisions to begin a garbage collection are based not on *need*, but on a policy determining when collections are most appropriate. In particular, my algorithms perform a collection after a specified number of bytes have been allocated (the collection *threshold*).

By thinking about two measures of garbage collection performance—collection duration and frequency—I can show how a threshold-based policy is preferable to a fixed-size policy. Consider the two parameters that determine the frequency and duration of collection: object birth rate and death rate. Assume constant rates over a number of collections. When the birth rate equals the death rate, the system is in equilibrium, and the two collection policies show the same behavior. However, if births outnumber deaths, while the frequency of collection using a threshold policy remains constant, the fixed space policy causes collections with increasing frequency, resulting in potential thrashing. Over the lifespan of a program, object births and deaths are generally equal, but if the semispace size is too small, there is a high probability that the rates of birth and death are very different, as illustrated in Figure 3.8. When the net allocation rate varies widely, threshold-based collection policies are more stable than fixed-size semispace policies.

Collections in each generation can be triggered by thresholds for that generation if promotions to a generation are counted against the generation's threshold. The threshold parameters (one per generation) have a tremendous impact on the performance of a system, and are one of the major parameters of interest throughout this thesis. Threshold sizes determine the locality, frequency, and duration of collections and, indirectly, the rates of promotion.

4.1.3 Traversing Reachable Objects

All garbage collection algorithms identify reachable objects by starting from a special set of pointers (the root set) and proceeding transitively to each reachable object. The root set includes the machine registers, runtime stack, binding stack, global variables, and intergenerational pointers if generation collection is performed. Because the root sets are similar for every algorithm, there is little variation between algorithms in traversing the root set.

Before discussing the different traversal policies for copying and non-copying collectors, I will mention the actions that must be taken by any traversal algorithm. In traversing reachable objects, any algorithm will visit each object and read every pointer in that object. Furthermore, some action must be taken to indicate the object has been visited (the mark operation) and each object must be tested when it is visited to determine if it has been marked.

Whatever algorithm is used, every pointer in a reachable object must be tested for several things. If the pointer is actually an immediate value, it is ignored. If the pointer points to a generation that is not being collected, it is ignored. Finally if a pointer indicates an object in the current generation, the mark for the object must be accessed to determine if the object has been traversed. Thus, there are many similarities in the algorithms that traverse reachable objects.

The major difference in traversal policies between copying and non-copying collectors arise from the recursive nature of traversal. Because objects contain more than one pointer, a record must be maintained to indicate which pointers of an object have been traversed. For example, as one pointer in a cons cell is followed and objects reachable from it are identified, the other pointer must be stored somewhere so the algorithm can eventually follow it as well. In a non-copying algorithm, the record is maintained by placing the pointers yet to be followed on a stack. Implementations of the stack vary, but the minimum cost of maintaining the stack is one push and one pop operation per object traversed.

Copying collectors use tospace as a list of pointers yet to be traversed. Two addresses in tospace are maintained: **copy**, which indicates where the next object in fromspace should be copied, and **scan**, which indicates the current address being scanned. Addresses between the base of tospace and **scan** have been scanned and the objects pointed to by pointers in this range have been copied. Pointers in addresses between **scan** and **copy** have yet to be traversed. When **scan** equals **copy**, all objects in fromspace have been transported.

The other difference between copying and non-copying algorithms lies in the way in which objects are marked as visited. Non-copying algorithms only need a single bit to indicate if an object has been visited. A bitmap can be used for this purpose. Because copying algorithms must also relocate pointers to the objects that have been transported, the new address of the object must also be retained somewhere. This *forwarding pointer* is typically stored with the old copy of the object in such a way as to indicate that the object has been transported. Another cost in copying collectors not associated with non-copying collectors is the updating of pointers to transported objects. In 1977, Clark and

Green showed that most objects have a reference count of one [19], indicating the cost of relocating pointers to transported objects is approximately one store per object transported.

4.1.4 Preserving Reachable Objects

I have just outlined how a copying algorithm preserves reachable objects by transporting them to tospace. Non-copying algorithms do not copy reachable objects, but instead mark them and then collect the unreachable objects. Objects are non-reachable if their mark bit is not set during the mark phase. As mentioned, clearing the mark bits and scanning for all unmarked objects requires time proportional to the size of memory. However, by combining generational collection with a bitmap representation of the mark bits, the execution time for maintaining the mark bits and sweeping the table can be made reasonable.

Generational collection allows a small fragment of the total heap to be marked and swept independently. By representing mark bits in a bitmap, the memory needed to maintain marks for a generation is compressed by a factor of 64 (1 bit per minimum sized object, an 8-byte cons cell). For generation sizes of 1–2 megabytes, the mark table is relatively small (64 kilobytes or less), and the cost of sweeping the mark table is not prohibitive.

A further enhancement to mark-and-sweep collection can improve interactive response. Instead of reclaiming unmarked objects immediately after the mark phase, the sweep phase can be deferred and performed incrementally. Sweeping can be tied to allocation, as in incremental copying collection, so that, for example, every 30 allocations the mark bits are checked and 30 more objects are collected. The noticeable pauses of mark-and-deferred-sweep collection can be tied to the mark phase alone.

4.1.5 Promotion

Policies for promotion interact closely with heap organization policies. This section discusses policies intended to prevent premature promotion, what can be done if premature promotion occurs, and how promotion policies interact with a non-copying generation algorithm, such as mark-and-deferred-sweep collection.

A collection algorithm benefits from promoting a long-lived object as soon as possible so that the object is not copied back and forth in newspace unnecessarily. Approximate information about object age can be maintained with the objects (copy count promotion), with a group of objects (Shaw's bucket brigade), or with the entire generation (en masse promotion). In my simulations, I explore the two extremes: copy count and en masse promotion. I believe maintaining per-group counts is probably the most effective strategy, but investigating the extremes provides the broadest range of information.

Data type is used in generation collectors as a basis for making decisions about promotion. Function objects, known to be mostly long-lived, are often allocated in areas separate from other types and never collected. Ungar and Jackson indicate that segregating bitmaps and large strings reduces premature tenuring in Smalltalk [85]. Ungar and Jackson also propose a sophisticated mechanism they call "demographic feedback" to further reduce premature promotion. This policy uses information about space size and rate of copying to prevent promotion when the prevention does not interfere with interactive response. Unfortunately, the allocation rates and object lifespans in Lisp are significantly different from those in Smalltalk. Algorithms with two generations, such as generation scavenging, do not allow the flexibility required to efficiently manage data allocated at a high rate. Chapter 7 looks at the rates of promotion observed for the three algorithms and investigates what policies can effectively manage data allocated at the rates expected for Lisp running on a high performance workstation.

One clear solution to the problem of premature promotion is to provide multiple generations and collect the intermediate generations as they fill. The greatest problem with multiple generations is designing them so that the collection of older generations is infrequent, or if frequent, then at least non-disruptive. Incremental collection provides the most attractive solution to the problem of disruptive collection of older generations. One of the goals of this thesis is to investigate incremental collection and determine if it is appropriate for this purpose. If incremental collection is not cost-effective, then exploring policies to allow non-disruptive collection of intermediate generations will be even more important.

A final aspect of promotion relates to implementing non-copying mark-and-sweep collection with generations. Although generation collection algorithms where objects never move can be envisioned, this thesis investigates a non-copying mark-and-sweep algorithm where objects are relocated when they are promoted. Unfortunately, because grouping objects by age is not feasible with a non-copying collector (since they never move and are reallocated as they die), either approximate lifespan information must be kept with each object, or the entire generation must be promoted together. The space overhead of maintaining a 1–4 byte copy count with each cons cell is unattractive. As an alternative, a count bitmap can be used, where several bits per object are used to maintain a small copy count per object. This approach trades off the space of a larger count with the execution time required to extract two to three bits from a word. Because I am interested in understanding the overhead of en masse promotion, and because it is the simplest policy to use in a non-copying collector, I investigate a mark-and-sweep algorithm that uses en masse promotion.

4.2 The Algorithms Simulated

The following chapters of this thesis present a comparative performance evaluation of three garbage collection algorithms: simple stop-and-copy collection, incremental copying collection, and mark-and-deferred-sweep collection. All algorithms are enhanced to include four generations. I have mentioned many of the policies I am exploring earlier in this chapter. This section provides an overview of each algorithm. In Appendix B, a pseudo-code implementation of each algorithm is provided.

4.2.1 Stop-and-Copy Collection

This is a simple copying algorithm augmented with four generations. Because of its simplicity and ease of implementation on stock hardware, this algorithm provides baseline data that the other algorithms can be compared against. If the performance of this algorithm is adequate, the more complex mechanisms of incremental and mark-and-deferred-sweep collection become unnecessary.

The algorithm is very simple. Each generation is broken into two extensible semispaces. A threshold of bytes allocated or promoted triggers a garbage collection. Copy count promotion is the promotion policy used. The scanning algorithm is exactly as described in section 4.1.3, implemented in pseudocode in Appendix B, and depicted as a flow diagram in Appendix A.

4.2.2 Incremental Collection

This is a variation of the previous stop-and-copy algorithm with enhancements to make it incremental. By making the two algorithms as similar as possible in most respects, the variations in performance directly related to incremental collection can be determined. The generations in incremental collection are organized just as in stop-and-copy collection, except that each semispace contains a separate region where newly allocated objects are stored. New objects are segregated from the areas being incrementally scanned because the new objects are known to contain pointers into tospace, and do not need to be scanned. Objects are promoted just as for stop-and-copy collection and a flip is invoked based on an allocation threshold, except when the flip is delayed because tospace has not been fully evacuated.

Scanning is performed in a manner similar to stop-and-copy collection, except that tospace is scanned incrementally—k words are scanned at every object allocation. The parameter k controls how rapidly fromspace is evacuated. When semispaces have a fixed size, k must be carefully chosen so that fromspace is evacuated before tospace is exhausted. If I assume flexibility in the size of tospace, that is, if I allow tospace to grow as needed, then the parameter k can be chosen to provide the best performance. Large values of k force fromspace to be evacuated quickly and cause incremental collection to perform more like stop-and-copy collection. Small values of k (< 1) distribute the process of evacuation more widely over the allocations and cause more objects to be faulted across the read barrier. By setting k to 0 for a period of time, Courts reports that the locality reference in the generation is significantly improved because objects are clustered together in tospace as they are referenced [25]. In this thesis, I assume that $k = 4$, the value suggested by Baker in the original paper on incremental collection.

4.2.3 Mark-and-Deferred-Sweep Collection

The final algorithm considered, mark-and-deferred-sweep collection with four generations, has not been previously implemented in any Lisp system. The closest available implementation is the garbage collector found in Kyoto Common Lisp, which also uses a non-relocating mark-and-sweep approach [93]. My algorithm enhances that one by adding a mark bitmap, generations, and deferred sweeping. The mark-and-deferred-sweep algorithm is much more complicated than either incremental or stop-and-copy collection because it avoids transporting objects until promotion. To accommodate vectors and types with variable size, it allocates headers separately from the variable-sized body of such objects. All pointers to the object point to the header, which does not move until it is promoted. The bodies of such objects are allocated in a separate two semispace region, and are collected and compacted as necessary. Since all pointers point to the non-moving headers of these objects, their bodies can be transported without relocating any pointers.

Because the mark-and-deferred-sweep algorithm is the only mark-and-sweep algorithm and the only non-copying algorithm considered in this thesis, the terms mark-and-sweep collection algorithm and non-copying algorithm will be used henceforth to unambiguously refer to the algorithm described in this section.

Each generation in the mark-and-sweep algorithm contains a bitmap, a fixed-size object region and a variable-sized object region (the relocatable region). The fixed-size region is divided into *areas*. Each area contains the objects of a single type. Thus, a one kilobyte area might contain 128 two-word cons cells, 51 five-word symbols, or 64 four-word vector headers. Initially, all areas in a generation are unallocated and linked together in a list of free areas. As storage is needed, areas are allocated and assigned a type. A list of allocated areas is maintained for each type.

When a collection occurs, the bitmap for the generation is cleared and objects are marked using the algorithm in section 4.1.3. The bodies of vectors allocated in the relocatable region are also copied and compacted using the two semispaces. For each type, a short vector is used to note the locations of n free objects of that type. After n allocations of a type, the bitmaps for areas of that type are swept for n more objects. In this way, the bitmaps are swept incrementally.

The mark-and-sweep algorithm is complicated even further because occasionally objects must be promoted. Because promotion requires relocating pointers and transporting objects, it requires a copying collection algorithm as well as a mark-and-sweep algorithm. As mark-and-sweep collection is implemented, when a promotion is needed, all objects in a generation are transported to the next generation.

Because it is so complex, the mark-and-sweep algorithm must show significant performance advantages over stop-and-copy collection for it to be preferred. There are several possible advantages. Because objects are infrequently transported during collection and the mark phase of collection touches only reachable objects, I expect the reference locality of mark-and-sweep collection to be better than that of the copying algorithms. Second, because objects do not move during collection, for multiprocessor garbage collection the mark-and-sweep algorithm may require less synchronization between processors during collection.

Chapter 5

Garbage Collection CPU Costs

This chapter investigates the CPU costs of the garbage collection algorithms described in Chapter 4. CPU costs fall into three categories: the base cost, or the cost of traversing and preserving objects; reference costs, including cycles needed to maintain the read and write barriers; and representation costs, which are the costs of referencing a data representation convenient for garbage collection. The evaluation of these costs is based on a model in which events of different types are counted and assigned a fixed CPU cost (measured in processor cycles). Costs are reported as a percentage of additional time required to execute the test programs. This chapter does not consider the effects of reference locality on performance.

These investigations attempt to answer three questions: what implementation techniques are most effective for generation collection, what is the minimum overhead required to implement Baker-style incremental collection on stock hardware, and is the mark-and-sweep algorithm competitive in CPU costs with a simple copying algorithm. Because the newspace threshold size has the greatest affect on performance of the generation thresholds, references to "the threshold parameter or size" in this chapter implicitly refer to the newspace threshold unless otherwise noted. The threshold parameter strongly influences the pause length, reference locality, and promotion rate of algorithms, and so CPU overhead is evaluated for a range of threshold sizes.

Throughout this chapter, costs of basic algorithm operations are estimated in machine cycles. These estimates are based on the instruction sequences required by a RISC-style architecture such as SPUR, MIPS, or SPARC. These architectures have single load/store addressing operations, one-cycle instruction execution, and delayed transfers of control. Appendix A contains the SPARC instruction sequences that were used to estimate the operation costs in this chapter. Explanations of the sequences are also provided in the appendix.

5.1 Costs Based on Events

The following comparisons estimate CPU costs of different implementations based on a count of events. One important event is a heap memory reference, which is an effective unit of temporal measure (see Chapter 3). Measurements of SPUR Lisp programs (see Table 3.1) indicate heap references account for approximately 12% of total machine cycles (i.e., approximately eight cycles per heap reference). Simulation of eleven large C programs on the MIPS R3000 architecture, involving hundreds of millions of instructions, show a range from 4.7–23.2 cycles per heap reference, with a mean of 7.8 [80]. Ungar reports a ratio of 5.2–8.8 instructions per data reference (including stack references) with a mean of 5.9 for seven Smalltalk programs executing on SOAR, a RISC processor with register windows [86]. If stack references are not counted, the mean ratio would again be close to eight instructions per heap reference. Based on this evidence from several sources, I assign a fixed cost of eight cycles to a program heap reference. Total user time (in cycles) is then measured as $T_{user} = E_{user} C_{user}$, where E_{user} are the number of user program references and $C_{user} = 8$.

The base costs of garbage collection can be measured by breaking down each algorithm into a flow graph of basic operations and assigning each operation a cost in cycles. When the algorithm is simulated, a count of the number of occurrences of each basic operation is maintained and by multiplying the cycles per occurrence by the number of occurrences, the total cycles per operation can be determined. The cost of collection, T_{gcbase}, is the sum of the costs of the basic operations. The cycle counts and operation flow graphs used are described in Appendix A. With the base cost of collection known, the overhead of collection is calculated as:

$$O_{gcbase} = T_{gcbase} / T_{user}$$

While this cost model does make simplifying assumptions, this chapter investigates comparative performance (not absolute) and estimates are sufficient for accurate comparisons because the simplifying assumptions apply equally to all algorithms compared.

Beyond the references necessary for program execution and collection, certain reference events require additional cycles. Stores of pointers into objects require that action is taken when an intergenerational pointer is created. Similarly, pointer loads require checks to implement the read barrier. Figure 5.1 indicates the relative frequency of different kinds of references for the four test programs (not including garbage collection references).

The five types of references indicated are: loadp, a load of a pointer; load, the load of an unboxed word; storep, the store of a pointer; store, the store of an unboxed word, and storei, a store to initialize an allocated object. The graph shows that pointer loads are the most common memory reference operation. Stores of all types are uncommon, except stores to initialize objects. Stores of pointers that can cause write barrier traps are quite infrequent ($<$ 10% in all cases). Stores to initialize (storei) can never cause a write barrier trap because a newly created object cannot have a generation older than the pointer being stored in it.

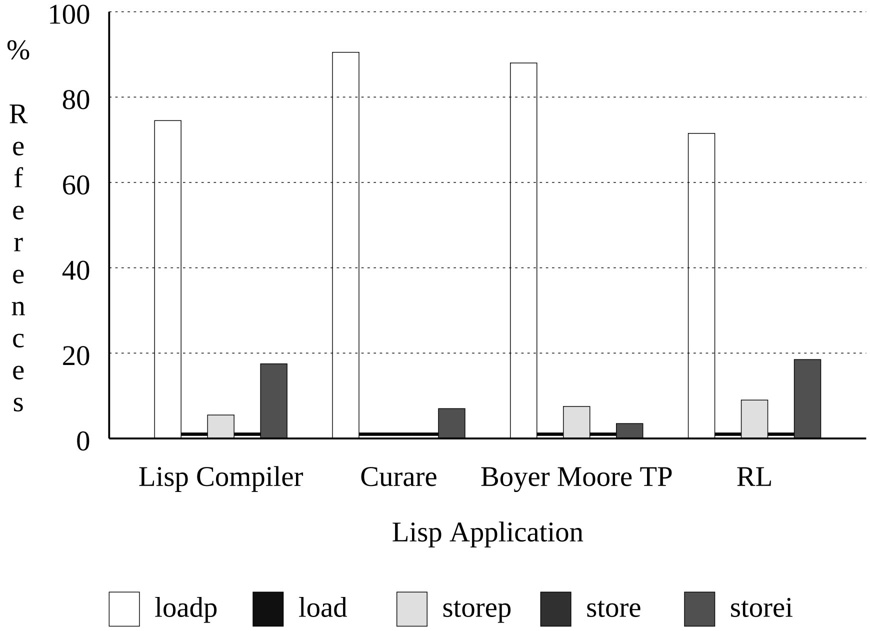

Figure 5.1: Object References by Instruction Type for Test Programs.

5.2 Implementing the Write Barrier

This section investigates possible implementations of the write barrier. For the collection algorithms considered, only pointers forward in time (from older generations to younger generations) need to be recorded. Maintaining the write barrier requires that action is taken whenever a pointer to a younger generation is stored into an object in an older generation. This event is called a *write barrier trap*. The cost of a write barrier trap can be simply modeled as:

$$T_{writebarrier} = E_{storep}C_{storep} + E_{wbtrap}C_{wbtrap}$$

where $T_{writebarrier}$ represents the total cycles required to maintain the write barrier, E_{storep} and E_{wbtrap} are the number of pointer stores and write barrier traps, respectively, and C_{storep} and C_{wbtrap} are the costs of these events in cycles. In all cases, $E_{wbtrap} \leq E_{storep} \leq E_{user}$. The ratios $R_{storep} = E_{storep}/E_{user}$ and $R_{wbtrap} = E_{wbtrap}/E_{user}$ determine how costly each pointer store and write barrier trap can be before they dominate the execution time of the program. Taylor reports values of 1–20% for R_{storep} and 0.0–0.64% for R_{wbtrap} assuming a newspace semispace size of 500 kilobytes [81]. Taylor fails to distinguish between pointer stores to initialize and pointer stores to update, hence his results are unnecessarily conservative. My results, summarized in Table 5.1, indicate R_{storep} ranges from 2–8% for larger programs.

Event Ratio	ACLC	Curare	BMTP	RL
R_{storep} (%)	5.6	2.4	7.7	8.1
R_{wbtrap} (%)	0.007	0.002	0.010	0.190

Table 5.1: Pointer Stores and Write Barrier Trap Ratios. Write barrier trap ratios were measured with a 512 kilobyte threshold size assuming stop-and-copy garbage collection with copy count promotion after three copies.

For a newspace threshold of 500 kilobytes, write barrier traps occur with less frequency than is reported by Taylor. Taylor's largest value for R_{wbtrap}, 0.64%, was measured for the FFT benchmark, which stores newly allocated floating point numbers into a long-lived array that is quickly promoted to an older generation. Taylor's one large benchmark, RSIM, has $R_{wbtrap} = 0\%$. The values reported here (0.002–0.020%) are probably more indicative of large programs. R_{wbtrap} depends heavily on how quickly objects get promoted (which in turn depends on the collection threshold size). This section measures the write barrier trap overhead for a range of threshold sizes and considers three possible implementations of the write barrier.

A hardware implementation of the write barrier provides the best available performance. The Symbolics implements the write barrier in hardware and completely eliminates the overhead (i.e., $C_{storep} = C_{wbtrap} = 0$) [61]. The SPUR hardware eliminates the per-store overhead but includes overhead on the trap [81]. This chapter concentrates on RISC architectures and so considers trap hardware similar to that provided by SPUR, which attempts to only provide hardware that significantly improves performance.

The cost of a write barrier trap can be further subdivided: $C_{wbtrap} = C_{trap} + C_{wbhandler}$. The cost of getting back and forth from the trapping instruction to the trap handler, C_{trap}, depends on the hardware and operating system support for machine traps. $C_{wbhandler}$, the cost of servicing the trap once inside the hander, depends on what actions are required by the handler.

The trap cost depends on what hardware support is provided by the architecture and whether or not the trap requires a context switch to the operating system. Hardware-supported write barrier traps do not require a context switch to the kernel, since all actions taken can be performed in user mode. If an architecture provides hardware for the write barrier, it probably supports fast traps. Johnson describes modifications to the SPUR architecture that would significantly reduce trap handler overhead [47]. The sequence, presented in Appendix A, simply transfers control, sets up operands for the handler, and, when the handler is through, returns. This cost of the sequence is seven cycles ($C_{trap} = 7$).

The cost of handling a write barrier trap, $C_{wbhandler}$, depends on the representation used to record the locations of intergenerational pointers. Simple representations maintain a sequence of addresses (typically as a vector), where each trap adds the address of the intergenerational pointer to the sequence [3, 86, 96]. One problem with a sequence representation is that the same address can occur in the sequence many times. Such redundancy wastes space and also time when the sequence is scanned during garbage collection. The advantages of a sequence representation are simplicity of implementation and speed of trap handling. Sobalvarro describes an organization, called word marking, where a bitmap is used to indicate the memory locations of intergenerational pointers [75]. The write barrier trap handler simply sets a bit in the bitmap. This technique avoids the redundancy of a sequence representation at the memory cost of the bitmap. Another alternative, which Sobalvarro calls card marking, uses a bitmap to indicate if an intergeneration pointer is stored in a region of memory (the card). The Symbolics uses such a method where the card is a hardware page [61]. Card marking is less desirable than word marking because at collection time each marked card must be scanned to locate the intergenerational references.

Sobalvarro describes a ten instruction sequence for the MC68020 architecture that implements write barrier word marking. The same sequence in the SPARC architecture requires 16 instructions, due to the simpler instructions and lack of indirect addressing modes (see Appendix A). Because word marking avoids the redundancy of the sequence representation and because its handler is sufficiently fast, word marking is the assumed method for recording intergenerational pointers and $C_{wbhandler}$ is assumed to be 16 throughout this chapter. Combining the estimated costs of a hardware write barrier, the result is $C_{storep} = 0$ (the test for intergenerational pointers is done in parallel with the store) and $C_{wbtrap} = C_{trap}(7) + C_{wbhandler}(16) = 23$.

An alternative to a special hardware implementation of the write barrier is to perform software checks before every pointer store. The sequence of instructions required to perform the check depends on the encoding of generations. Possible encodings include generation tag bits stored with every pointer, as in SPUR, or maintaining knowledge about the address range that each generation occupies and comparing pointers with the ranges to determine inclusion. The address range check is more suitable for stock hardware because after encoding types in pointers there is no room left for generation tags.

The correct implementation of the write barrier software check combines a single fast inline test to handle the most frequent case (pointer stores into newspace) with a function call that performs the other tests. The cost model for this implementation is

$$T_{writebarrier} = E_{newstorep}C_{newstorep} + E_{oldstorep}C_{oldstorep} + E_{wbstores}C_{wbhandler}$$

where $E_{newstorep}$ is the number of pointer stores to newspace, $E_{oldstorep}$ is the number of pointer stores to oldspace, and $E_{wbstores}$ is the number of stores that cause a write barrier trap. The instruction sequence for this method can be found in Appendix A. This implementation avoids the code expansion of putting all the generation range checks (30 instructions) inline at every pointer store. If generations are designed so that newspace occupies higher (or lower) addresses that oldspace, then the inline test is just a single compare of the pointer being stored into against the base of newspace ($C_{newstorep} = 3$). The function that handles other cases must check for an immediate being stored and then look at the relative generations of the pointers. From the appendix, this cost, $C_{oldstorep}$, is estimated at 13 cycles. As with the hardware implementation, $C_{wbhandler}$ is 16 cycles.

A different implementation of the write barrier does not use special hardware, but instead uses the page protection mechanisms provided by most operating systems. This approach write-protects oldspace so that when a pointer is stored to a protected page, the operating system fault handler is used to handle the write barrier trap. Two variations of the basic approach are considered: one variation that is unacceptable and one that is attractive.

A naive implementation using operating system traps write-protects oldspace and forces an operating system trap on every pointer store into oldspace. The evaluation model is:

$$T_{writebarrier} = E_{oldstorep}C_{oldstorep} + E_{wbtrap}C_{wbtrap}$$

where $E_{oldstorep}$ is the number of pointer stores into oldspace and $C_{oldstorep}$ is the cost of the protection fault caused by the store. $C_{oldstorep}$ does not include the cost of handling the write barrier trap, since that cost is included in C_{wbtrap}. Similarly, C_{wbtrap} does not include the fault overhead, and so $C_{wbtrap} = 16$, the cost of the handler.

The effectiveness of this implementation depends on two factors: what fraction of total pointer stores store into oldspace and how fast protection faults can be serviced by the operating system. The relatively small fraction of write barrier traps observed (0.002–0.020%) suggests that pointer stores of any type into oldspace are infrequent. The only additional pointer stores that would cause traps, if all stores to oldspace trapped, would be stores of oldspace pointers into oldspace objects. Surprisingly, the frequency of oldspace pointer stores into oldspace is quite high (possibly caused by stores that are binding the values of special variables). Table 5.2 indicates the fraction of total pointer stores that are stores of pointers into oldspace for the four test applications. Contrasted with the small

Ratio	ACLC	Curare	BMTP	RL
$E_{oldstorep}/E_{storep}$ (%)	9.0	5.0	61.7	12.7

Table 5.2: Fraction of Pointer Stores into Objects in Oldspace.

fraction of stores causing write barrier traps, we see that up to half of all pointer stores can be to oldspace objects. Fortunately, most of these stores do not cause write barrier traps that need to be handled. The high frequency of such stores makes the cost of an implementation that traps on every pointer store into oldspace unacceptable (overheads in the 100's of percent).

Instead of trapping on every pointer store into oldspace, a more clever implementation will only trap once per oldspace page that is stored into. If the oldspace pages written between garbage collections are correctly noted, they can be scanned for intergenerational pointers at the time of collection. Shaw suggests that small modifications to the virtual-memory user interface of traditional operating systems would allow this information to be maintained very cheaply [74]. Without operating system changes, the collection algorithm must maintain its own dirty-bit information, at the cost of one protection fault per oldspace page written.

Using this approach, after a garbage collection all pages in oldspace are write-protected and the dirty bits are cleared. Between collections, if a page in oldspace is written, the dirty bit for that page is set and the page is made writable. When the next garbage collection occurs, all the dirty pages are scanned for intergenerational pointers. Each intergenerational pointer discovered is recorded (using a sequence or word-marking) so that its exact location is known during subsequent collections. The cost of this approach is estimated with the following model:

$$T_{writebarrier} = E_{makedirty}(C_{makedirty} + C_{scanpage})$$

where $E_{makedirty}$ is the number of initial pointer writes to oldspace pages, $C_{makedirty}$ is the cost of handling the write protection fault and marking the page as dirty, and $C_{scanpage}$ is the cost of scanning a page for intergenerational pointers. $C_{scanpage}$ requires scanning a page containing 1024 words, looking for newspace pointers. The per-word cost of scanning is five cycles, resulting in $C_{scanpage} = 5120$. The cost of writing an oldspace page can also be broken down: $C_{makedirty} = C_{opsys} + C_{handler}$. The handler that marks a page as dirty can be very fast, so $C_{handler}$ is small, perhaps five instructions. The other source of cost is the operating system.

The operating system must do two things when a protected page is written. First, it must detect the fault and transfer control to the user fault handler. Second, the user will ask the operating system to change the protection of the page, making it writable. The cost of fielding a protection fault depends on whether the fault handler is executed by the kernel or not. Because protection faults are typically initially captured by the kernel for security reasons, if the handler is executed in user mode, it requires two context switches (from user to kernel and back). Context switching is very machine dependent, but always requires instructions to save and restore state. In the Sprite operating system on the SPUR and SPARC architectures, a context switch requires an absolute minimum of 70 instructions [7]. If a fault is handled in the kernel, the switch to the fault handler requires a minimum of 35 instructions.

On top of the context switch overhead, the write barrier trap must make the protected page writable. This action is also very architecture dependent, since protection changes may invalidate data stored in a virtually-addressed cache. If the cache requires flushing, as it will with the SPUR and SPARC architectures, many instructions may be required (256–512 stores, depending on page size).

Another estimate of the cost of a protection fault is found in an analysis of the cost of faults used to implement a copy-on-write policy for process forking. In the Sprite operating system, Nelson estimates a copy-on-write fault in the SPUR architecture would require 500 instructions to implement with the handler executing entirely in the kernel [63]. Nelson supplies a similar estimate for the Mach operating system copy-on-write protection fault.

The conclusion to reach from this discussion is that the operating system cost of the write barrier trap will be at least a couple of hundred instructions, and possibly as many as one thousand. Because there is a range of possible costs, I consider two values for C_{opsys}. A well-designed operating system might have $C_{opsys} = 200$ and I call this implementation "fast OS trap". A typical operating system will not support fast protection faults (assuming they are infrequent and cause by program errors) and might have $C_{opsys} = 1000$ ("slow OS trap").

Figure 5.2 shows the predicted overhead of maintaining the write barrier as a function of threshold size using the three techniques suggested (and two possible operating system costs for protection faults). A stop-and-copy algorithm is used for these results, but the results for the incremental and mark-and-sweep algorithms are similar. Overhead decreases as threshold size increases because larger thresholds cause fewer collections and fewer objects are promoted. Since promoted objects cause write barrier traps when pointers to newspace objects are stored in them, fewer traps occur when fewer objects are promoted, hence the decrease in overhead. In the absence of other considerations, a larger threshold size is clearly better. The figure also shows that the operating system trap methods are much more sensitive to threshold size than the software or hardware methods. This sensitivity is caused in part by the high operating system overhead (200–1000 cycles), but mostly by the high cost of scanning a page (5120 cycles).

Assuming threshold sizes larger than 125 kilobytes, we see that the write barrier implemented with protection faults is faster than a software test implementation, even with slow operating system traps. For large threshold sizes, where very little data gets promoted, we see that the protection fault implementation is sometimes comparable to the hardware implementation, whose overhead is negligible in all cases. The protection fault implementation does not show great sensitivity to the cost of the operating system trap, and so does not require fast traps because the page-scanning overhead dominates the cost of this method.

While the cost of a software test implementation is also relatively small in most cases, the protection fault implementation appears more effective for larger thresholds. The major disadvantage of the protection fault implementation is that it requires cooperation from the operating system, which must allow the user to set protections for regions of memory and handle protection violations in these regions. Commercial vendors of Lisp systems, who port their product to many different systems and architectures, may find the increased portability of software tests outweighs its additional overhead.

5.3 Implementing the Read Barrier

Because incremental collection avoids pauses associated with collecting large spaces, it is an attractive alternative to stop-and-copy collection. While hardware has been used to minimize the overhead of incremental collection, Baker-style incremental collection has never been implemented cost-effectively on stock hardware. This section discusses possible implementations of incremental collection on stock hardware and provides cost models for their performance.

As with the write barrier cost, the read barrier cost is modeled by assigning costs to

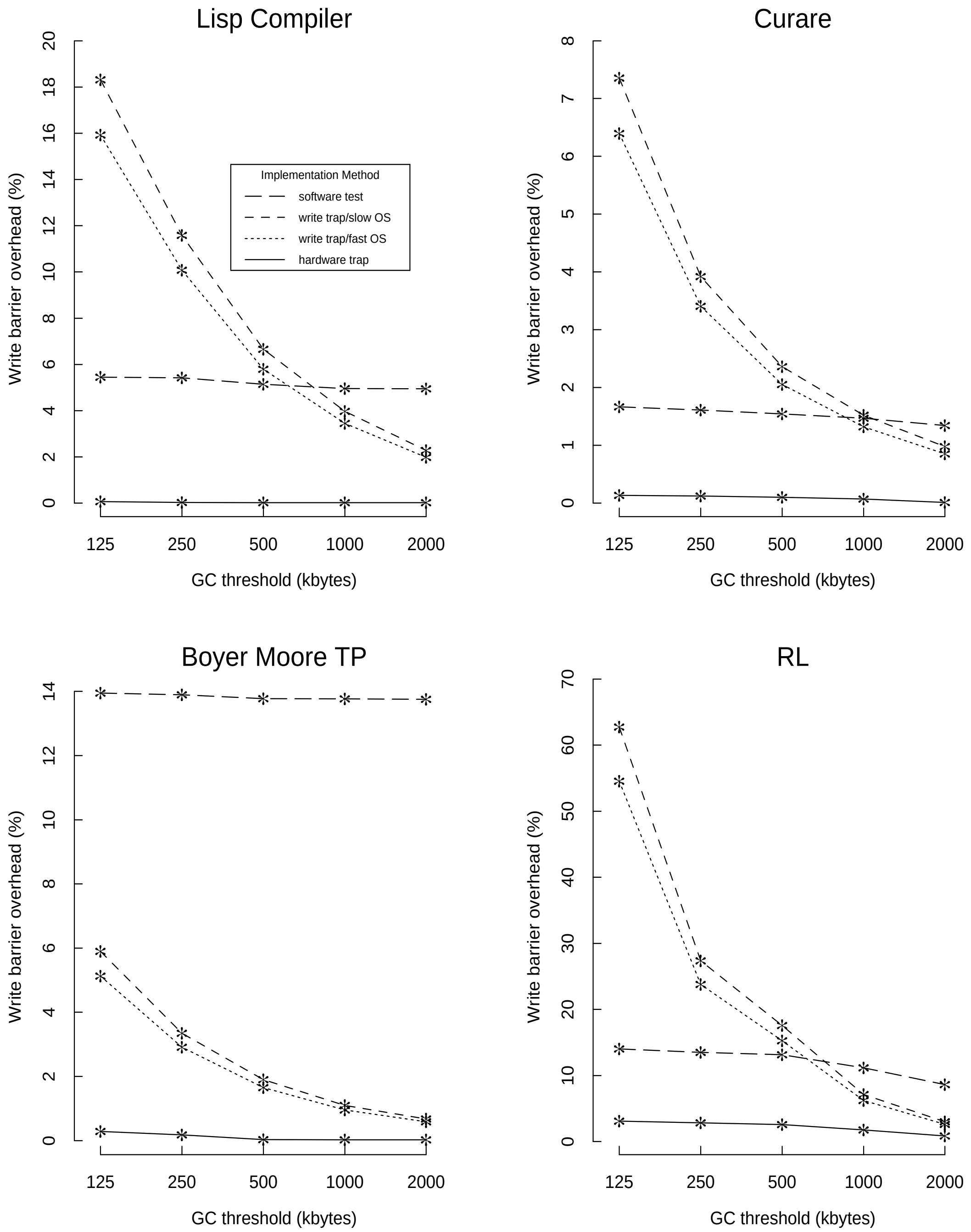

Figure 5.2: CPU Overhead for Write Barrier Implementations. Stop-and-copy is the garbage collection algorithm used in all cases.

events. The read barrier requires that every load of a pointer checks if the loaded pointer points into fromspace. A simple cost model for the read barrier is:

$$T_{readbarrier} = E_{loadp}C_{loadp} + E_{rbtrap}C_{rbtrap}$$

where E_{loadp} is the number of pointer loads, and E_{rbtrap} is the number of loads of a pointer into fromspace.

With a hardware implementation, every load of a pointer performs a hardware check to determine if the loaded pointer points into fromspace. One would expect a pipelined architecture could perform the check in parallel with executing subsequent instructions, and hence completely eliminate the overhead of the test. But when the hardware test traps it must prevent any subsequent instruction from completing (to prevent copies of the loaded pointer from propagating). In the best case, the instruction following a pointer load is constrained not to manipulate the loaded pointer. In the worst case, the load takes an extra cycle. If we assume the worst case ($C_{loadp} = 1$), we will see that its cost is not large.

Every pointer load that traps must transfer control to a handler that transparently relocates the object in fromspace. As with the write barrier traps, the overhead of the trap depends on the architecture, but in the best case requires approximately seven cycles. Finally, the relocation of the object requires cycles, but these cycles are not counted as a cost of the read barrier, since they must be performed by a copying collector anyway. For a hardware read barrier, $C_{loadp} = 1$ and $C_{rbtrap} = 7$.

Software tests can also be used to implement the read barrier. As with the write barrier, an inline check is combined with a function that handles pointers into fromspace. Appendix A contains the instruction sequences for the test and the handler. The one-comparison test possible for the write barrier will not work for the read barrier because flipping changes the relative positions of fromspace and tospace in memory. Depending on the pointer, a fromspace test takes an additional three or six cycles. The average, $C_{loadp} = 4.5$, is assumed. This implementation disallows multiple generations from being collected simultaneously because tests for inclusion in multiple fromspaces would be required at each load, significantly increasing C_{loadp}. When a fromspace pointer is found, there is little overhead other than the call to and return from the handler function, and $C_{rbtrap} = 9.5$.

Because so many of the memory references performed by a program are pointer loads (see Figure 5.1), a software implementation of the read barrier introduces a large overhead. As alternatives to inline checks on stock hardware, two read barrier implementations that use operating system traps to maintain the read barrier should be considered. These methods remove the invariant that fromspace pointers are not allowed in machine registers (which is what the read barrier is designed to prevent). Instead of checking for fromspace pointers as they are loaded, the pages of fromspace are read and write protected, and objects are transported only when the memory they contain is referenced and causes an operating system trap. When the trap occurs, the object is relocated into tospace and the pointer to it is updated. Because fromspace pointers are allowed in registers, a copy of a fromspace pointer can be created before its object is referenced and relocated. Such object "aliases" must be either recognized or removed. Furthermore, aliases must be prevented from being stored in memory. Both operating system trap implementations of the read barrier add checks around pointer stores to prevent fromspace pointers from being stored in memory. Inline checks around pointer stores (including initializing stores) similar to those around pointer loads in the software implementation accomplish this goal (with a cost of three cycles).

There are two implementation alternatives if fromspace aliases are allowed. The first alternative allows aliases to exist in registers, but must insure that aliases are recognized as being the same object. The eq operation, which tests for identical pointers, is modified to consider aliases as follows. First, the pointers are checked for being identical. If they are identical, no further action is needed because no aliases were involved. If the test fails then there are two possibilities. First, if one argument is known to be nil or a fixnum immediate (as generated by the compiler), then the test fails and no aliases were involved because nil and immediates cannot have aliases.[1] The only case where aliases need be be checked occurs when both arguments are not recognized by the compiler and the eq test fails. In this case, the pointers must be checked for being aliases to each other. Table 5.3 indicates the relative frequency of eq tests and heap references and the frequency of outcomes of eq tests for several large test programs.

Operation	RSIM	Weaver	SLC	PMA	Average
Eq true, total (%)	52.1	36.6	27.3	36.2	38.1
Eq false, total (%)	47.9	63.4	72.7	63.8	61.9
Eq false, with nil (%)	24.6	27.4	41.4	31.0	31.1
Eq false, with immediate (%)	8.6	25.1	0.8	7.8	10.6
Eq false, 2 unknown args (%)	14.8	10.8	30.6	25.0	20.3
Eq tests / heap access (%)	11.4	42.0	36.9	19.7	27.5

Table 5.3: Relative Frequency of Outcomes of EQ Tests. RSIM, Weaver, and PMA are described in Appendix A. SLC is the SPUR Lisp compiler.

The table indicates that while a majority of eq tests fail, only a relatively small fraction of the tests (10–30%) would require tests for object aliases. Appendix A shows the modified instruction sequence for eq tests if fromspace aliases are allowed. The added cost of false eq tests with two unknown arguments is 12 cycles. The total cost model of implementing the read barrier with aliases and a modified eq test is:

$$T_{readbarrier} = E_{eqfalse2unk} C_{eqfalse2unk} + E_{allstorep} C_{allstorep} + E_{rbtrap} C_{rbtrap}$$

where $C_{eqfalse2unk} = 12$, the cost of a false eq test with two unknown arguments, $C_{allstorep} = 3$ and $C_{rbtrap} = C_{opsys} + C_{handler}$. $C_{handler} = 0$ because the relocation is required anyway.

[1]Nil is stored in the oldest generation and not incrementally transported.

As with the write barrier, C_{opsys} varies widely depending on the architecture and operating system. Again, two costs for C_{opsys}, 200 cycles (fast OS trap), and 1000 cycles (slow OS trap), are considered (see the discussion in Section 5.2).

An alternative to allowing aliases to appear freely in the stack and machine registers is to remove all fromspace aliases when an object is relocated. This method requires scanning the stack and registers at relocation time and avoids any special actions on eq tests. Unfortunately, the cost of scanning the stack and registers varies greatly with architecture. Furthermore, the cost is strongly related to the average depth of the stack, which is very application-dependent. I have measured the average stack depth for several large Lisp programs and the results are presented in Table 5.4.

Metric	RSIM	Weaver	SLC	PMA	Average
Stack depth (frames)	7.0	33.4	25.2	21.6	21.8
Average register usage	3.8	3.4	6.5	6.6	5.1

Table 5.4: Average Stack Depth for Several Common Lisp Programs. The frames measured are the register window frames in the SPUR architecture. RSIM, PMA, and Weaver are described in Appendix A. SLC is the SPUR Lisp compiler.

One influence machine architecture has on stack scanning is the register model of the architecture. Register windows, as defined in SPUR and SPARC, allow fast function calls but assign a fixed minimum number of registers per stack frame (the register window size). A general register model, as found in the MIPS architecture, allows compilers to allocate stack frames containing only as many registers as are needed. Register windows increase the cost of stack scanning because the average register usage (3.8–6.6 registers) is significantly smaller than the typical register window size (16). This section estimates the cost of stack scanning with and without register windows, and then uses the register-window estimate for comparison with other methods because it is the more conservative estimate.

Using the register-window model, the cost of scanning a stack frame can be estimated as the register window size times the average stack depth. The cost of stack scanning is estimated as:

$$C_{stackscan} = C_{scan} N_{framesize} D_{avgstackdepth}$$

where $C_{stackscan}$ is the cost of stack scanning, C_{scan} is the number of cycles to scan one stack element, and $N_{framesize}$ is the number of registers per register window and $D_{avgstackdepth}$ is the average stack depth of the application (in frames). For both SPUR and SPARC, $N_{framesize} = 16$. C_{scan} is roughly 6, counting loop overhead and the test for a fromspace pointer. Using the average from the applications in the table, the average stack depth is assumed to be 22 frames. Thus, the cost of scanning a register-window stack is estimated as 2112 cycles. Assuming a general register model, $N_{framesize}$ can be estimated from the table as 5.1. This reduces the scanning cost to 673 cycles. Nevertheless, the conservative estimate is used because the scanning costs are very application dependent, and should not be underestimated.

Having estimated the cost of stack scanning, the total cost of implementing the read barrier with stack scanning can be modeled as:

$$T_{readbarrier} = E_{allstorep}C_{allstorep} + E_{rbtrap}C_{rbtrap}$$

As before, $C_{allstorep} = 4$ and $C_{rbtrap} = C_{opsys} + C_{stackscan}$. We have just estimated that $C_{stackscan}$ with register windows would average 2112 cycles. For the operating system trap overhead (C_{opsys}), two alternatives, 200 cycles and 1000 cycles, are considered.

Overheads of four implementations for the read barrier are presented in Figure 5.3. Both slow and fast operating system (OS) trap implementations are considered for the alias and scanning methods. As with the write barrier, the read barrier overhead decreases as threshold size increases. Fewer objects are faulted across the barrier when fewer flips occur. The hardware implementation adds 9–11% to the cost of execution and the software implementation adds 40–50% overhead, both implementations being relatively insensitive to threshold size. Methods using OS traps to support the read barrier show a huge range of overheads (7–700%). These methods are only effective with large threshold sizes which minimize the number of objects copied across the barrier. The method that allows aliases and modifies eq tests appears to be the best of the OS trap implementations. Even with slow OS traps, this approach performs better than software checks in many cases, and significantly better in some (four times better for the Boyer Moore theorem prover). For a two megabyte threshold, the overhead of the modified eq method (with slow OS traps) varies from 13–63%.

Even with a conservative estimate, the stack scanning method with slow OS traps also shows better performance than software checks in some cases. Surprisingly, this approach performs better than the hardware implementation in the theorem prover. Unfortunately, this approach is very sensitive to stack depth, which can vary greatly between applications. For the applications measured, the stack scanning approach appears effective, but because it does not offer significant performance improvements over the modified eq method and is more sensitive to applications, the modified eq method should be preferred.

In any event, the overhead of incremental collection without hardware support is significant (10–50%), but not necessarily unacceptable. Furthermore, using OS traps to implement incremental collection is contingent on large threshold sizes. The additional read barrier faults caused by small threshold sizes add tremendous overhead in all OS trap implementations.

5.4 Base Costs of Garbage Collection

After having looked at the overhead of various implementations of the read and write barrier, this section compares the base costs of different collection algorithms. This section

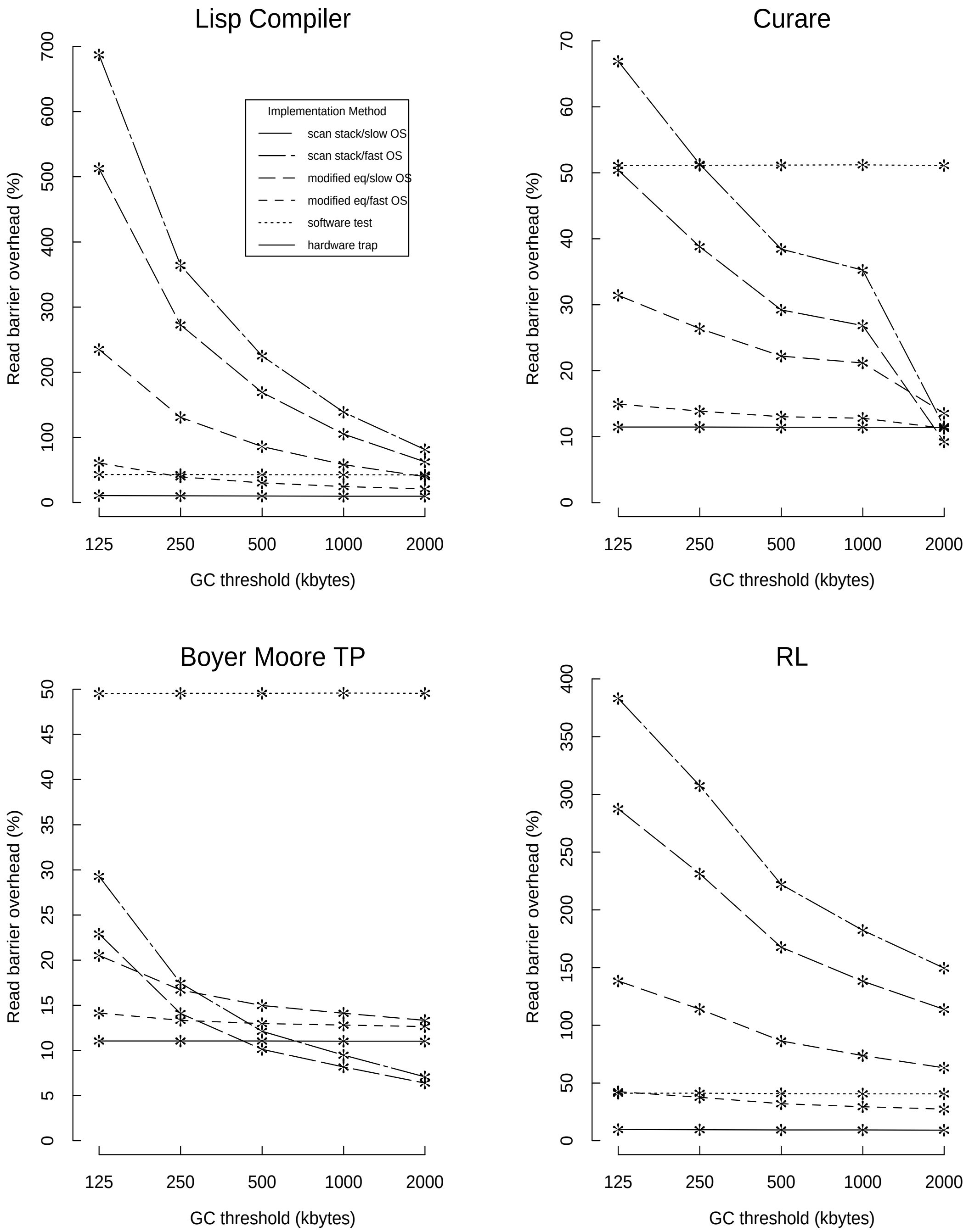

Figure 5.3: CPU Overhead for Read Barrier Implementations. The results indicate the overhead of the read barrier for an incremental copying algorithm.

investigates whether the cost of mark-and-sweep collection is competitive with the costs of the more traditional copying techniques.

Figure 5.4 illustrates the different parts of the overhead for a copying collector as a function of threshold size. The costs are broken down into allocating, scanning, forwarding, transporting, and updating. The allocation cost is independent of threshold size and ranges from 1.5–5.3%. Small variations in the allocation overhead result from interactions between MARS and the host Lisp system. The scanning cost includes dereferencing and testing each word in tospace looking for fromspace pointers. Scanning is a large part (about 50%) of the non-allocation cost of copying collection. The forwarding cost, a small part of the total, includes type determination and checks for forwarding pointers. The transport cost, the second largest source of copying collection overhead, includes cycles to determine if an object needs promotion and memory references to transport the object. Finally, the update cost includes cycles to install forwarding pointers and relocate pointers to transported objects. Because the two largest sources of overhead, scanning and transporting, are necessary in any copying algorithm, it seems unlikely that the base cost of copying collection can be reduced significantly below the overhead measured.

The figure also indicates how the copying overhead varies with threshold size. While the total overhead varies by an order of magnitude between applications, the overall relationship between threshold size and overhead is similar for all test programs. Large thresholds (two megabytes) provide approximately a factor of three improvement over small thresholds (128 kilobytes). Overhead for large thresholds is small (<20%). Because the allocation cost is independent of threshold size, it provides a lower bound on the total overhead associated with copying collection.

Next, we look at the observed overhead in a mark-and-sweep collector. Figure 5.5 shows the cumulative overhead for the four test programs. This overhead is broken into six parts: allocating, sweeping, stack operations, type checks, marking, and vector transports. As with copying collection, the cost of allocation is independent of threshold size. Non-copying allocation is slightly more costly than copying allocation, especially for vector objects. The sweeping cost includes the overhead of sweeping the mark bitmap, and is also largely independent of threshold size (i.e., approximately one bit swept per object allocated in a sparsely-marked bitmap). Sweeping is slightly less expensive than allocating, but because it is also independent of threshold size, sweeping raises the minimum overhead achievable with very large thresholds. Still, this minimum overhead is relatively small, from 1.7–12.1% (compared with 1.2–5.3% in copying collection).

On top of sweep costs, the costs of object marking are roughly evenly divided between stack operations (pushing and popping addresses), type and generation checking, and accessing the mark bitmap. Relocating vector bodies incurs minimal cost, especially when few vectors are allocated, as in the theorem prover. Even though bitmap testing is a relatively expensive operation with a RISC architecture (12 instructions in SPARC), the testing overhead is not prohibitive. The overhead is low because mark tests are only performed on newspace pointers, and there is approximately one newspace pointer per newspace object

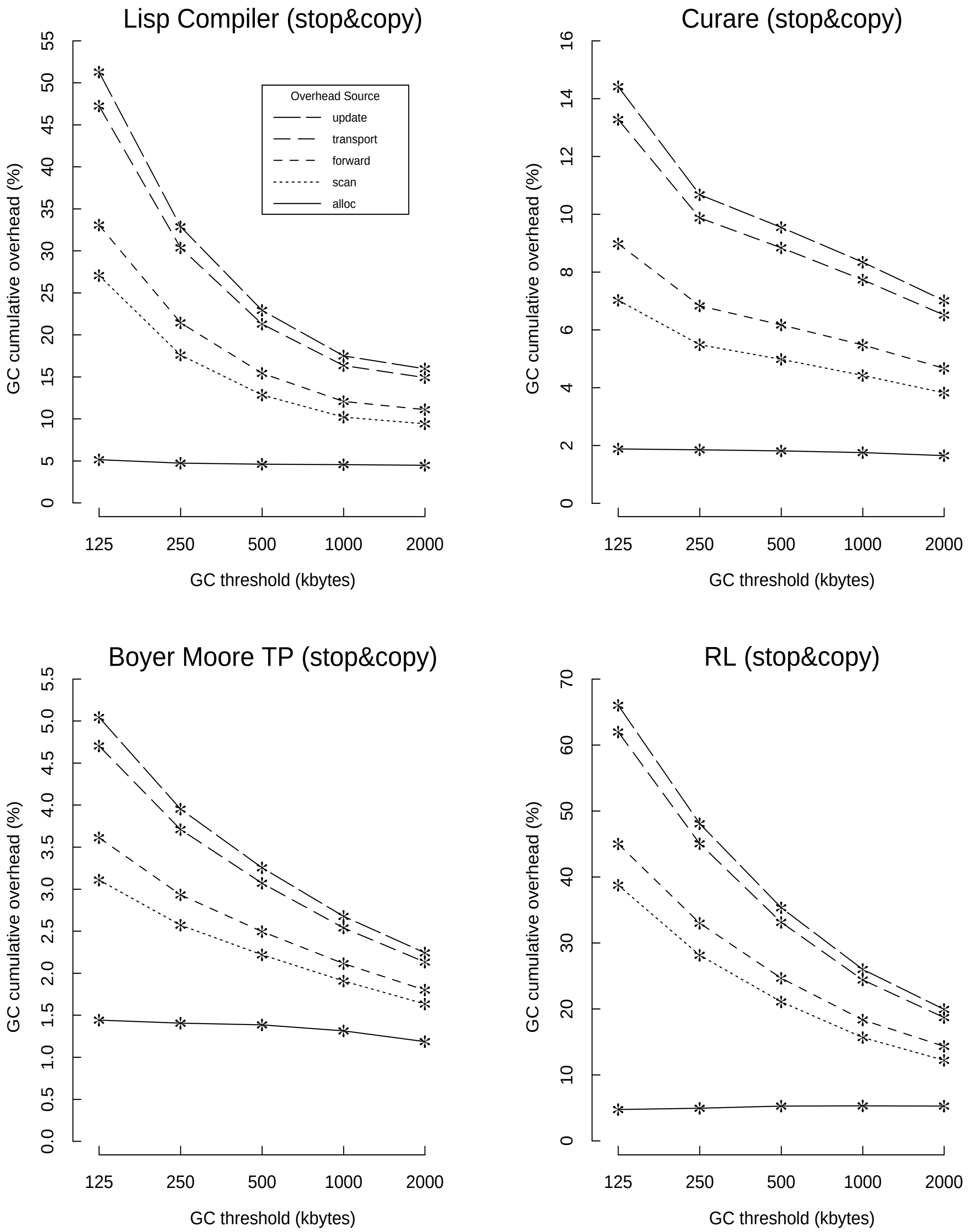

Figure 5.4: Cumulative CPU Overhead of Copying Collection. The algorithm used is stop-and-copy collection. Results for incremental copying are similar.

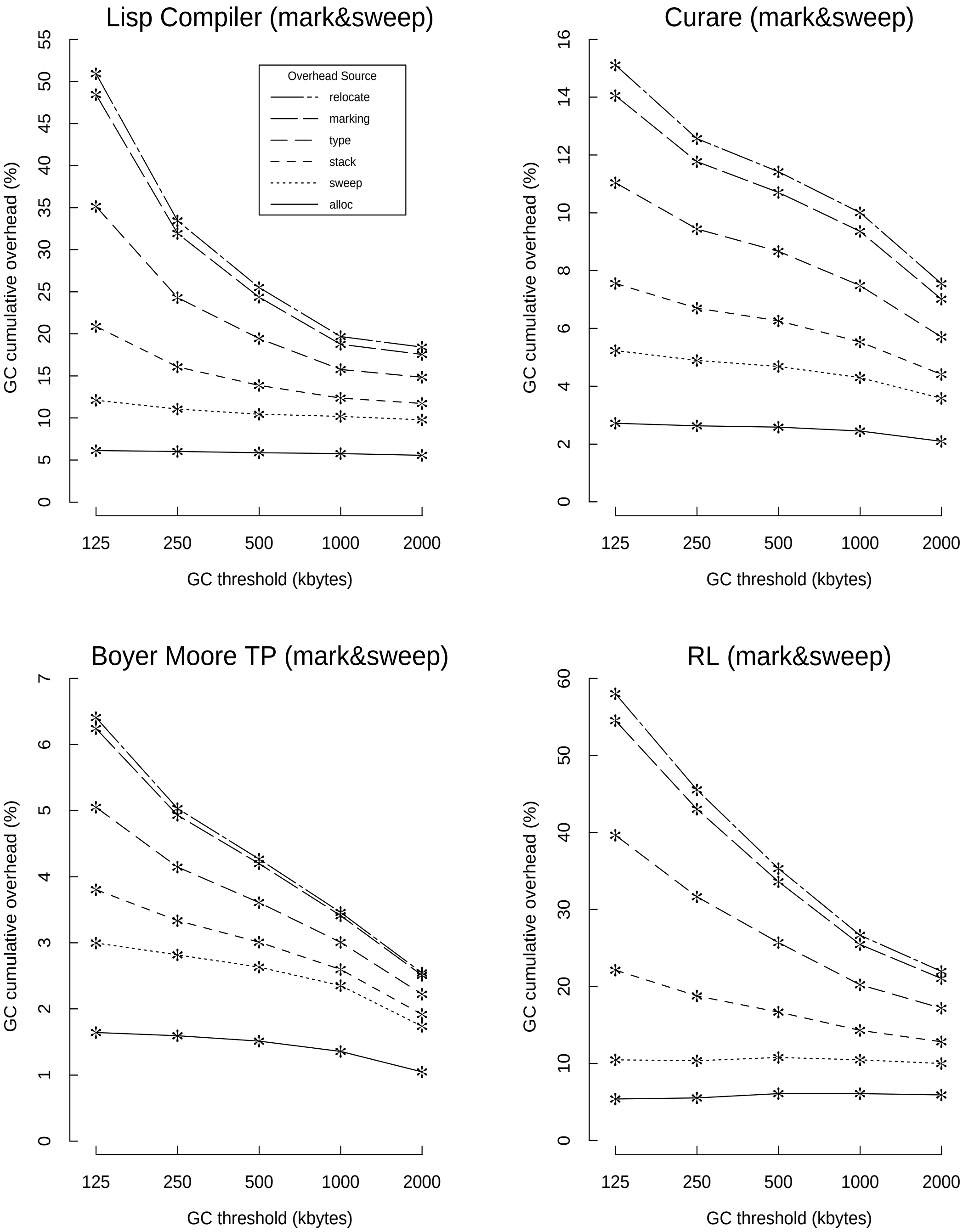

Figure 5.5: Cumulative CPU Overhead of Mark-and-Sweep Collection.

(i.e., approximately one mark test per object transported).

The total overhead of mark-and-sweep collection ranges from 2.5–58%. For small thresholds, mark-and-sweep collection has lower overhead than stop-and-copy collection, and for large thresholds the reverse is true. This turnabout occurs because the overhead of mark-and-sweep collection has two components: marking, which is very threshold dependent, and sweeping, which is threshold independent. Apparently, the mark phase alone is slightly faster than the entire copying algorithm. For small thresholds, the overhead from the threshold-dependent part of the algorithm dominates and mark-and-sweep is slightly faster. But as the threshold-dependent overhead decreases, the total overhead is dominated by the threshold-independent costs, which are higher for mark-and-sweep collection. Using bitmaps and deferred sweeping, the mark-and-sweep algorithm shows slightly worse performance than copying algorithms. While the mark-and-sweep algorithm has disadvantages (increased complexity and increased promotion rate), it also has definite advantages over copying collection, as the next chapter shows.

5.5 Total Overhead

In conclusion, the predicted overheads are combined to arrive at a total cost of collection for the three algorithms using the most cost-effective implementations for the read and write barriers. The write barrier is assumed to be implemented with oldspace write protection faults. The read barrier is assumed to be implemented with fromspace read protection faults and the method that allows fromspace aliases and modifies the eq test. In both cases, slow operating system faults are assumed, as they are likely to be found in current operating systems. Figure 5.6 shows the different parts of the total overhead for the three algorithms on two of the test applications.

For large threshold sizes, the write barrier adds about 10% to total overhead. The read barrier adds considerably more, from 30–60% to the overhead of incremental collection. Indirect vector references only add a small percentage to the overhead of mark-and-sweep collection. While stop-and-copy collection and mark-and-sweep collection have roughly comparable costs (with mark-and-sweep collection showing relatively increased overhead for larger thresholds), incremental collection on stock hardware is considerably more expensive, even with the cheapest implementation of the read barrier. With architectural and operating system support for fast traps, this overhead could be significantly reduced. For very low overhead incremental collection, small amounts of additional hardware appear necessary. This chapter has considered the CPU costs of garbage collection independent of the locality of reference. Very fast algorithms with bad reference locality can perform poorly. The next chapter investigates the effect of garbage collection on locality of reference.

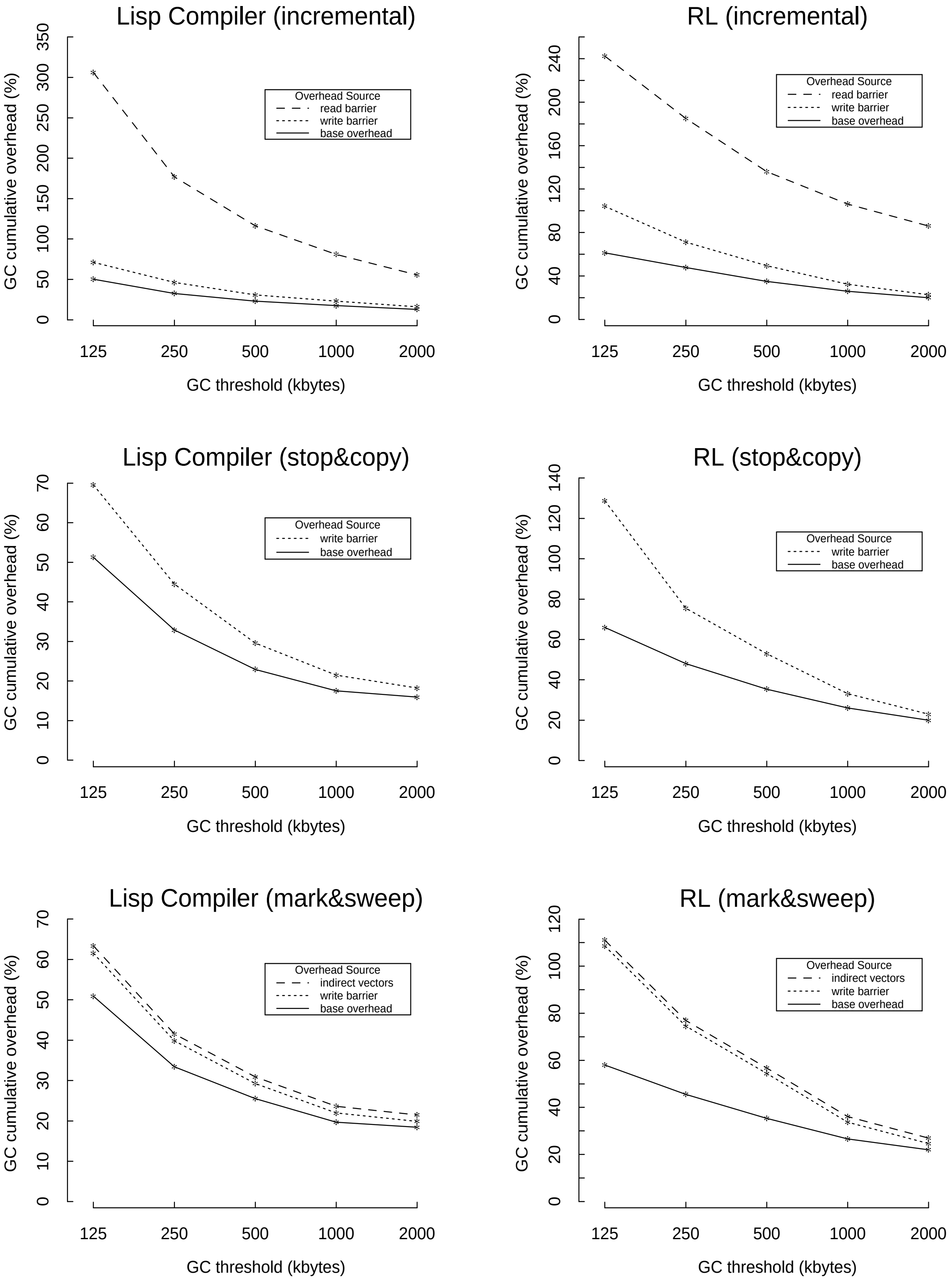

Figure 5.6: Cumulative CPU Overhead for Three Algorithms.

Chapter 6

Reference Locality

Locality is a property of the spatial and temporal pattern of a program's references to the memory. *Spatial locality* indicates how tightly grouped references are—references to addresses that are close together have high spatial locality. *Temporal locality* indicates the distribution of references in time. If the same group of objects is referenced repeatedly over a period of time, the temporal locality of these references is high. The degree of temporal and spatial locality of references determines the effectiveness of techniques devised to "buffer" or "cache" parts of a program's address space. Because we are most interested in the effectiveness of these caching techniques, the cache *miss rate* is an important measure of the combined spatial and temporal locality of reference in a program.

This chapter quantifies the strong influence that garbage collection algorithms have on a program's locality of reference. The first section describes the important characteristics of memory hierarchies and introduces the simulation techniques that are used to measure reference locality. The next section discusses the general reference characteristics of Lisp programs, and the final sections present locality results for the three collection algorithms being studied.

6.1 Memory Hierarchy Characteristics

Memory hierarchies have become complex and now often include on-chip instruction buffers and data caches, chip-to-board datapaths, board-level caches, memory buses, main memories, I/O channels, disk drives, and potentially larger secondary storage devices such as optical disks. This chapter focuses on how well programs fit in the board-level cache and main memory.

Each level (or slice) of a memory hierarchy has three basic parameters that determine its performance: capacity, latency, and bandwidth. The capacity of a slice depends on several factors, but most importantly on economics. In particular, faster memories are also more expensive. Bit for bit, cache memory is more expensive than main memory which in turn is more expensive than disk storage. In general, a user wants to minimize the cost of his system, and so prefers the minimum capacity that offers acceptable performance. Other capacity considerations include the amount of board space available for cache chips, the number of backplane slots for memory boards, or the amount of disk space available for a swap partition. Of the configuration parameters, capacity is the parameter that users can most readily control. My studies look at program performance over a variety of slice capacities to determine the cost/performance tradeoffs for different configurations and algorithms.

The latency of a slice is a parameter over which users have little or no control. Disk I/O latency has remained in the tens of milliseconds for more than a decade. The parameter that is changing, however, is the ratio of processor speed to latency. While disk latencies have remained unchanged, processors have increased in performance from 10–100 times and will continue to speed up. Additional CPU computation to avoid extra page faults becomes more and more cost-effective. Generation garbage collection provides the ideal mechanism to match the memory requirements of an application with the existing system's CPU to disk latency ratio. By decreasing the newspace threshold size, at a cost of increasing the frequency of garbage collection, any desired page fault rate can be achieved.

The interconnect bandwidth is another important characteristic of a slice. In uniprocessors, the memory bus and I/O channels are designed so that channel bandwidth is typically not a bottleneck in performance. But bandwidth is also dependent on existing technology, and although the technology exists to speed processors up by a factor of 100, or to add 100 processors to a memory bus, technologies that increase bus bandwidth by a factor of 100 do not exist yet. Therefore, in systems of the future, the bandwidth of the memory hierarchy will be increasingly important. Because interconnect bandwidth is most important in multiprocessor systems, further discussion of interconnect bandwidth will be postponed until Chapter 8.

More complex factors, such as cache and memory management policies and configuration also affect performance. Cache set-associativity affects cache cost and effectiveness. Cache block (or memory page) size and replacement policy also affect performance. Because there are so many variations, it impossible to look at every parameter in detail. To simplify the analysis and focus more on the effect of the garbage collection algorithm on performance, I assume certain common policies, such as a write-back cache and an LRU replacement policy. Furthermore, the cache block size is fixed at 32 bytes and the cache is assumed to be direct-mapped. These parameters closely resemble those of the SPUR system [42]. Trace-driven simulation research associated with the SPUR project concluded that the memory system parameters chosen were effective for a wide variety of test programs [44].

6.2 Measuring Miss Rates

Because similar techniques are used to measure the miss rate in the CPU cache and main memory, the term "cache" is often used to refer to both when discussing simulation techniques. In this section, a *cache* is any slice in the memory hierarchy used to buffer chunks of memory from the next larger slice. When talking specifically about the on-board CPU cache, I will use the term *CPU cache*.

The technique used to measure locality of reference in this thesis is *stack simulation*, first described by Mattson and others [59]. Stack simulation allows the miss rate for any number of cache sizes to be calculated with only one pass over the reference string. Because the hardware configuration and replacement policies differ between the CPU cache and main memory, the stack simulation techniques used to measure locality of reference also differ.

6.2.1 Stack Simulation

Stack simulation will only provide miss rate information if the replacement algorithm on a cache miss is one of a class of algorithms called *stack algorithms*. While not all replacement algorithms are stack algorithms (FIFO replacement is not), some of the most effective (such as LRU replacement) are. I will assume LRU replacement throughout this thesis. Stack simulation works because of the *inclusion* property of stack algorithms: all entries in a smaller cache will also be present in a larger cache. With this property, the simulation simply needs to determine, for every reference, the minimum cache size for which the reference will cause a hit. All smaller caches will miss and all larger caches will hit.

Stack simulation maintains the blocks referenced as a stack. With LRU replacement, when a block is referenced, the stack is searched. If the referenced block is located in the stack it is moved to the top of the stack, while the blocks now below it are shifted down one slot. The depth of the referenced block in the stack is the *stack distance* of the reference. If the block has never been referenced and is not found when the stack is searched, the reference has distance infinity and the new block is placed at the front of the stack. For a cache of size k blocks, all references with distance $k+1$ or greater will cause a miss. By maintaining a count of the stack distances derived from the reference string, the number of misses for any cache size can be determined with one pass over the reference string. A more comprehensive description of stack simulation is provided by Thompson [83].

6.2.2 Partial Stack Simulation

The stack simulation technique described above works well for determining page fault rates in main memories of any size. A straightforward implementation of the algorithm maintains the LRU stack as a linked-list of blocks which is repeatedly searched. The greatest drawback of stack simulation implemented in this way is that the stack distance must be computed for every one of tens of millions of references. Determining the distance of a block in a stack implemented as a linked-list of blocks requires time proportional to the average depth of the stack. With good reference locality, the average stack depth may be relatively small (Thompson reports a mean depth of 33 from an address trace simulation). But because garbage collection itself introduces significant non-locality, the mean stack depths I observe are larger (from 50 to 100).

Several techniques have been suggested to improve the efficiency of stack simulation. Bennet and Kruskal implemented stack simulation by creating a tree with size proportional to the length of the reference string [9]. Each node in the tree corresponds to a distinct reference to a block. Using their approach, the cost of computing the stack distance of a block reference is logarithmic in the distance. Olken extended their technique by reducing the space requirements [67]. Thompson investigated the performance of these algorithms and hybrid algorithms, which use different representations for different parts of the stack [83]. Interestingly, comparing the performance of the various methods, Thompson found the simple linked-list implementation was the most effective for address traces. Only when the mean stack depth became large (300–400) did the tree implementations show significantly better performance than the linked-list implementation.

In an effort to improve the performance of the linked-list implementation, I have modified it to perform faster at a cost of information loss. Recognizing that miss rate information about small caches is not interesting (few people will run Lisp on machines with 64 kilobytes of main memory), I modified the linked-list implementation by splitting the LRU stack into two parts. The new algorithm is called *partial stack simulation*. I define a parameter, N, and modify the algorithm so that stack distances smaller than N are not determined precisely. The elements of the stack are thus divided into those within the top N entries, and those below the top N entries.

The simple observation that makes this approach possible is that references to the top N blocks have no effect on blocks farther down in the stack. Only the relative positions of the top N entries changes. By eliminating the distance computation for blocks within the top N entries, many stack operations are avoided.

My partial stack simulation algorithm is formally defined in Figure 6.1. The pseudocode

---

function *referenceBlock(block)* **is**
    LastRefTime(block) $\longleftarrow$ *CurrentTime*
    if *(block* $\in$ *TopNSet)* **then**
        return *1*
    else
        distance $\longleftarrow$ N + *findStack(block, BlockStack)*
        removeStack(block, BlockStack)
        newBlock $\longleftarrow$ {*b* $\in$ *TopNSet* such that *LastRefTime(b)* is oldest}
        TopNSet $\longleftarrow$ *TopNSet* $-$ *newBlock* + *block*
        pushStack(newBlock, BlockStack)
        return *distance*
    endif
end

Figure 6.1: Partial Stack Simulation Pseudocode

used in the figure follows the conventions described in Appendix B. The *stack* data structure used in the figure, along with the normal *popStack* operation, supports *findStack*, which locates a block in a stack of blocks and returns its distance, and *removeStack*, which removes a block from a stack. Associated with each block is the attribute *LastRefTime*, which indicates when the block was last referenced. The global variables used in the algorithm are *TopNSet*, a set containing the N blocks that would traditionally be found at the top of the simulation stack, *BlockStack*, a stack of blocks containing the blocks found below the top N in the simulation stack, *CurrentTime*, an integer representing the current time, and N, a constant integer indicating how many elements are found in *TopNSet*. The $+$ and $-$ operations performed on a set denote the union and deletion operations, respectively.

The function *referenceBlock* takes a block identifier as an argument and returns the distance of the block in the simulation stack. Blocks within the top N entries of the traditional simulation stack are reported as having distance one. All other distances are correctly reported.

Using this approach, the parameter N determines the tradeoff between speed and information loss. When $N = 1$, the algorithm is just traditional stack simulation and the results are identical. As N increases, the miss rate of smaller caches is lost. Table 6.1 shows the performance of a simulation with N ranging from 1 to 250. For these simulations, $N = 250$

Stack Entries Ignored	$N = 1$	$N = 20$	$N = 50$	$N = 100$	$N = 250$
Execution Time (seconds)	177.9	126.3	109.1	103.7	98.2
Percent of full analysis	100.0	71.0	61.3	58.3	55.2
Speedup over full analysis	1.0	1.4	1.6	1.7	1.8
Percent of total list operations	—	62	48	35	21

Table 6.1: Effectiveness of Partial Stack Simulation. The execution time is the amount of time spent doing partial stack simulation for one of the test programs (the Lisp compiler). Percent of total operations indicates the fraction of total list cells that needed to be scanned with partial analysis.

corresponds to a main memory size of one megabyte, the minimum memory size of interest. As the table shows, increasing N to 250 speeds up the stack distance calculation by almost a factor of two. For $N = 250$ only 20% as many stack elements are scanned as for full analysis, but the simulation still requires 55% of the time. This discrepency is explained by the fixed costs per reference, which include function call and bookkeeping overhead (e.g., recording the current reference time for each reference). Almost doubling the speed of the stack distance computation, partial stack simulation is a simple technique that significantly decreases the overhead of stack simulation.

6.2.3 Measuring CPU Cache Miss Rates

The measurements of the CPU cache miss ratio for a variety of cache sizes (and potentially different associativities) were computed using a cache simulator, Tycho, written by Mark Hill [43]. Using a technique called all-associativity simulation, Tycho can determine miss ratios for a number of cache sizes and associativities with one pass over the reference string. A complete description of the technique and its performance is provided in Hill's thesis [44].

All-associativity simulation makes certain assumptions about the cache. There must be a single cache block size for all caches simulated (in this case, 32 bytes). Associative caches must use LRU block replacement. Furthermore, block prefetching and cache sub-blocks are not allowed.

6.3 Locality of Garbage Collected Programs

In a program that employs garbage collection, there will be two distinct patterns of reference: those of the program itself and those of the collection algorithm. To help distinguish between the different phases of execution, *program locality* refers to the locality of the user program, *collector locality* refers to the reference locality during garbage collection, and *overall locality* refers to the combined locality of the program and collector. This section discusses the reference patterns one would expect to see with a program running a generation-based garbage collection algorithm. Surprisingly, much can be said about the patterns of reference without considering specific applications.

The executing program references three distinct classes of objects: temporary objects allocated as intermediates for short computations, longer-lived program objects that survive a significant fraction of the entire computation, and Lisp runtime system objects (e.g., system packages and symbols) that have indefinite lifespans. A well-tuned generation garbage collection algorithm that promotes longer-lived objects will locate each class of objects in a different generation. The pattern and frequency of reference to each class (and generation) is distinctly different.

Program references to the newest generation include initializing stores. As we saw in Chapter 5, initializing stores account for a large fraction of all writes into the heap. Furthermore, initializing stores have the property that the previous contents of the allocated memory is known to be uninteresting. Peng and Sohi suggest a special "allocate" cache operation that tells the cache not to bother fetching the block being allocated since its contents will immediately be overwritten, avoiding unnecessary bus traffic [68]. Such an operation would also be helpful to the operating system when zeroing or copying virtual memory pages, actions that are common when a parent process forks a child process. The allocate operation would also be helpful to a copying collector, where copying proceeds sequentially through tospace. The allocate cache operation would be less useful for non-copying algorithms, where only parts of cache blocks are typically allocated. Other program references to the newest generation are less predictable, although if the generation is kept relatively small a certain amount of spatial program locality is guaranteed.

References to the older generations are mostly loads, as most stores are initializing stores. Because older program objects get compacted as they are promoted, references to these objects should show good spatial locality. Premature promotion of data that dies quickly will dilute the spatial locality of references to the second generation and should be avoided. In the following sections, we will see the negative effect of premature promotion. System objects, on the other hand, are often widely distributed in the oldest generation, which is typically megabytes in size. Reorganizing these objects for better locality can be effective in improving program locality, especially if the reorganization is intended for the execution of a specific set of applications. Courts' training-band approach shows how successful the reorganization of system objects can be [25].

In summary, program references fall into two very distinct categories: newspace references, which are local and include predictable stores to initialize, and oldspace references, which are more diffuse and include few stores. Figure 6.2 presents the frequency distribution of the ages of objects referenced for the four test programs. In the figure, we see that the age distribution is bimodal. There are many references to very young objects, but there are also a large number of references to old objects. References to system objects, which account for 18–28% of all references, are not shown. This result suggests that even if the entire newspace can be placed in the cache or main memory, references to older generations can significantly reduce the program locality. One solution, as explored in this chapter, is to make caches and main memories large enough to include both newspace and the active objects in oldspace.

By its nature, poor collector locality is somewhat unavoidable because garbage collection algorithms must reference every reachable object in the generations being collected. This pattern of reference shows both poor spatial and temporal locality. Beyond these necessary references, copying and non-copying algorithms make additional references.

Copying algorithms transport every object, resulting in a series of allocate operations in tospace. They must also write into the fromspace copy of objects to mark the forwarding address and update pointers to relocated objects. Furthermore, the tospace copies of objects must be scanned. Essentially, copying algorithms require reads and writes to every object in fromspace and two sequential writing scans (first to copy and then to relocate) of objects transported to tospace. The advantage of copying algorithms is that the reachable objects are compacted as they are copied and newspace spatial reference locality should be improved. If, however, newspace fits entirely in the cache or memory, this effect is not significant.

The order of traversal in a copying algorithm affects the locality of garbage collection, as discussed by Moon [61]. Courts' reports that approximate depth-first traversal was shown to decrease page fault rates by 10–15% over breadth-first traversal in the TI Explorer [25].

Among copying algorithms, incremental algorithms reference memory in a very different pattern from stop-and-copy algorithms. Instead of separating program and collector references into distinct groups, incremental collection finely intermingles collector and program

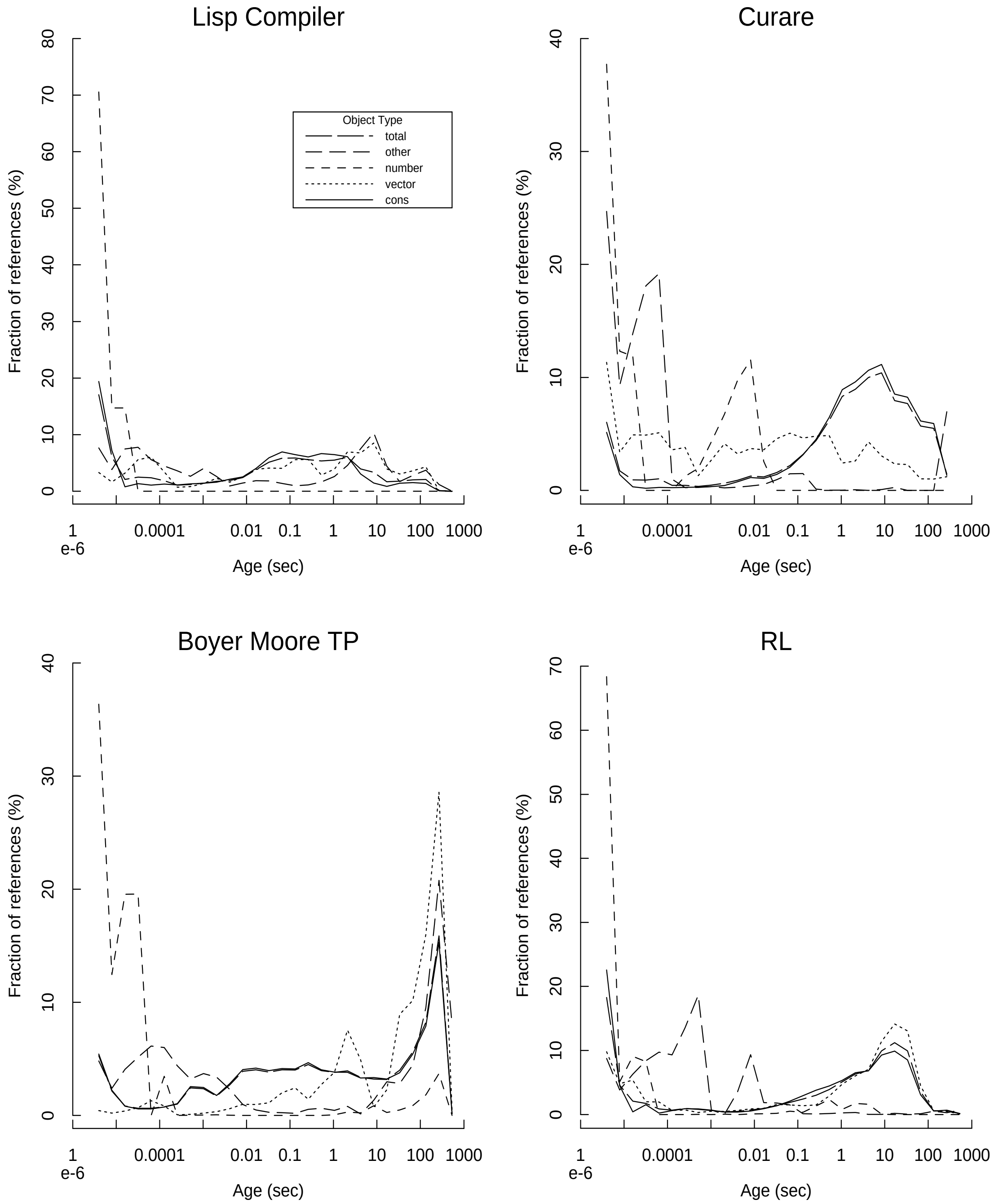

Figure 6.2: Age Distribution of Objects Referenced by Object Type.

references. The result should be a decrease in overall locality because of interference between the two kinds of references. Collector references dilute the program locality and vice versa. The degree of this dilution is measured later in this chapter.

Non-copying algorithms, such as mark-and-sweep collection, have significantly better collector locality than copying algorithms. In particular, non-heap references are made to a stack and a mark bitmap, both data structures that have good locality characteristics. Furthermore, objects in the heap are not modified at all. Intuition suggests that if the entire newspace fits in the cache, non-copying collection will require a smaller cache than copying collection to achieve the same miss rate.

6.4 Main Memory Locality

This section investigates the locality of reference in the main memory for the three garbage collection algorithms described in Chapter 4. The primary measure of locality in the main memory is the page-fault rate, the number of page faults generated by a program in a second. Since page faults require 10–30 milliseconds to handle, page fault rates on the order of 10 faults per second are the highest tolerable rates (adding 10–30% to the execution time of the program). This section examines the advantages and disadvantages of each algorithm over a variety of space sizes and estimates each algorithm's memory requirements. Of greatest interest is the relative locality of incremental and stop-and-copy collection (intuition suggests that stop-and-copy should have better locality), and the relative locality of copying and non-copying collection (intuition suggests non-copying should have better locality). Before presenting the results, I discuss the limitations of a simulation-based approach for measuring main memory locality.

Generation garbage collection greatly improves the locality of reference for all the algorithms measured. In fact, eight megabytes of physical memory is sufficient to insure that none of the test programs will cause a page fault even with a two megabyte newspace threshold. This result suggests that main-memory locality is unimportant for programs using generation garbage collection. Historically, however, Lisp programs have quickly grown to use all the physical memory that is available. Even existing programs challenge the current relatively large physical memories. I have been informed of an existing Lisp application for CAD that has a 200-megabyte working set [48].

The results presented must also be considered with this caveat in mind. I assume that the allocation and reference behavior of larger and longer running programs is similar to that of the programs I have measured so that the memory sizes considered and page fault rates will scale accordingly. Furthermore, my results only apply to the performance of a single program. Another source of paging not considered here results from transitions between programs, causing large changes in the working set. To investigate the VM performance under these conditions, interactive workloads would have to be captured and fed into MARS.

6.4.1 Page Fault Rates

The page fault rates (faults per second) for the three collection algorithms are presented in Figure 6.3, where memory sizes from three to six megabytes are considered. The figure shows results from the two programs that required the most memory, the Lisp compiler and RL. The figure shows that as the newspace threshold increases, the fault rate also generally increases. Memory configurations that show acceptable performance are much larger than the newspace threshold. For the copying algorithms, the page fault rate tends to remain acceptable while the physical memory is about two times the threshold size plus 2–3 additional megabytes to contain oldspace data. For non-copying collection, a good "fit" results when the memory is as large as the threshold size plus 2–3 megabytes. From this we see that the non-copying algorithm shows much better reference locality than the copying algorithms. In particular, thresholds up to 500 kilobytes produce acceptable fault rates in a three megabyte memory for the mark-and-sweep algorithm, but not for the copying algorithms. Mark-and-sweep collection shows lower fault rates in almost all cases, except for the smallest threshold size with smaller memories. In these cases, the additional promotion caused by en masse promotion increases mark-and-sweep references to the second generation and dilutes reference locality.

Comparing stop-and-copy with incremental collection, the two methods appear to have similar fault rates. The interleaving of collector and program references appears to have little effect on reference locality in most cases. For small threshold sizes, where more objects are being incrementally transported from fromspace, the stop-and-copy algorithm does show better locality. Interestingly, for large thresholds, incremental collection shows slightly lower fault rates than stop-and-copy collection. This difference may be due to a slightly decreased rate of copying caused by the transport delay of incremental collection, which gives objects slightly longer to die, and hence transports fewer objects.

The figure shows a clear advantage for mark-and-sweep collection. With a smaller newspace size (i.e., no semispaces), more oldspace pages can permanently reside in memory, reducing the fault rate of this algorithm. Semispace compaction, which is intended to improve reference locality, actually decreases locality by increasing the size of newspace.

6.4.2 Memory Requirements

Another way to understand the main-memory locality of our algorithms is to determine the amount of physical memory required to achieve a particular fault rate as a function of algorithm and threshold size. Figure 6.4 shows the physical memory requirements of the algorithms for fault rates ranging from zero to twenty faults per second. Interestingly, the relationship between memory needed and threshold size is relatively flat, where one would expect to see the memory requirements steadily increasing with threshold size. There are conflicting forces that tend to flatten the relationship between threshold size and the memory needed. As threshold size increases, fewer collections occur and less data is promoted, causing fewer second generation references, thus decreasing the demand for memory in the

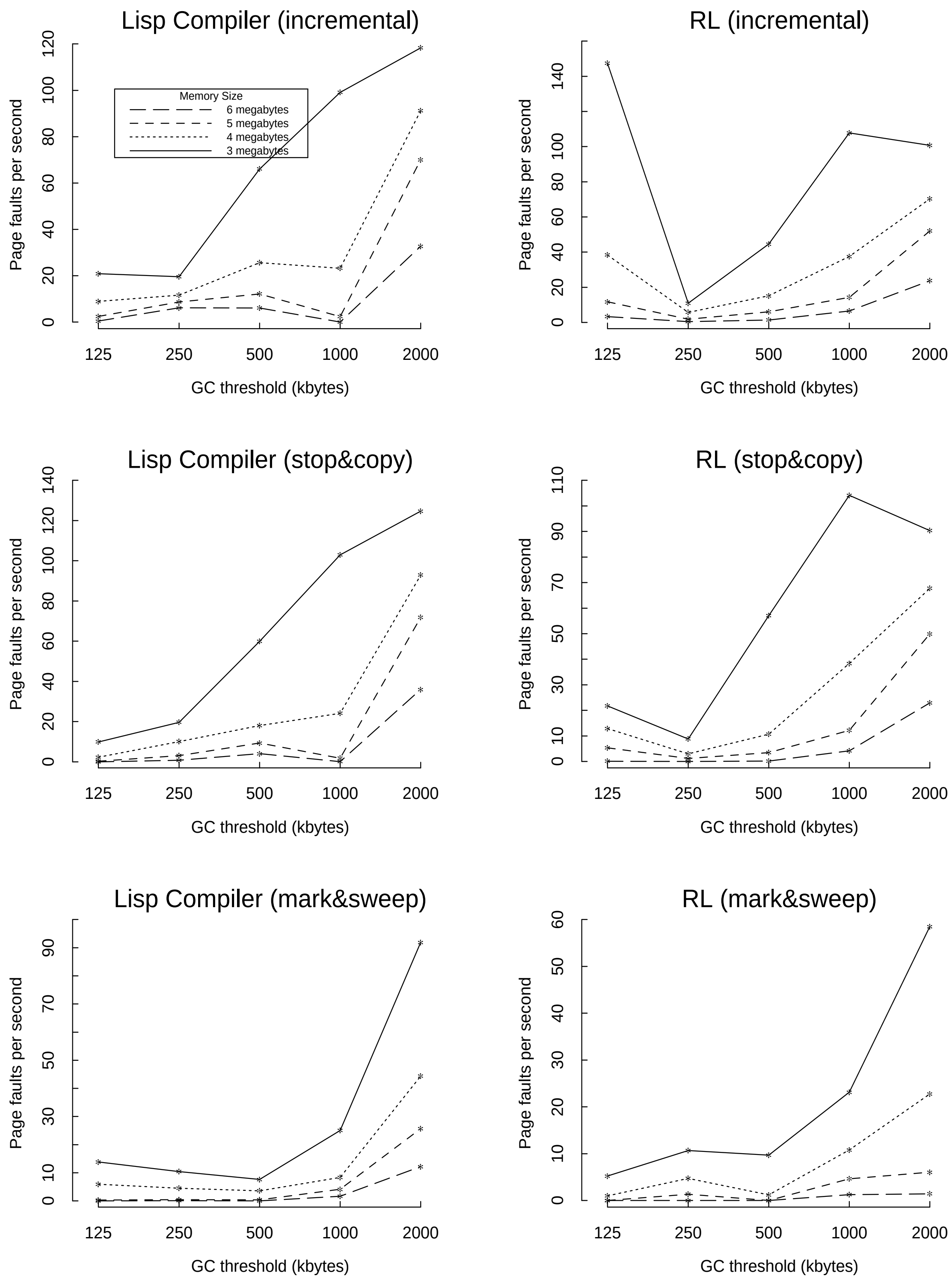

Figure 6.3: Page Fault Rates for Different Collection Algorithms.

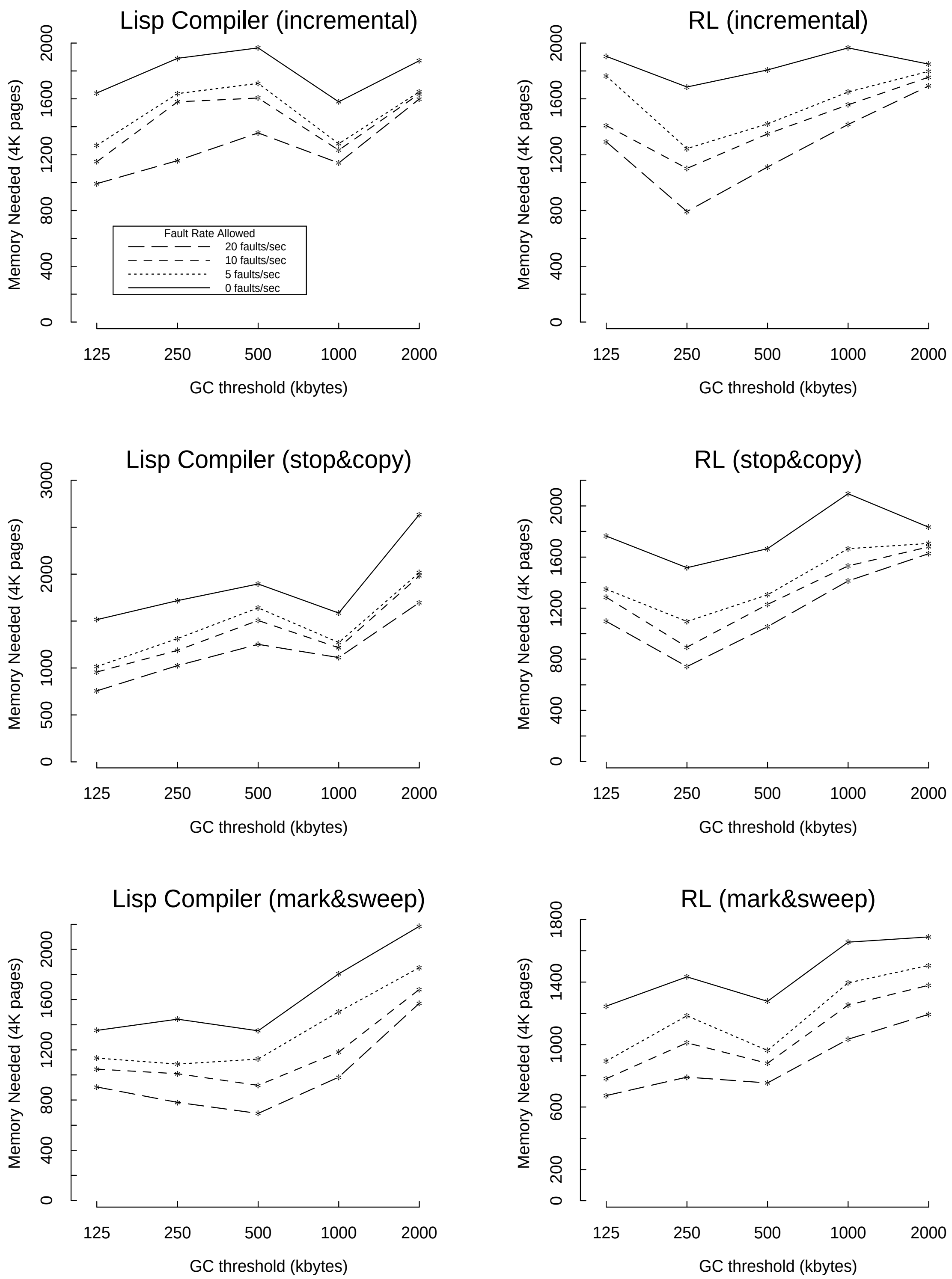

Figure 6.4: Memory Sizes Required for Different Collection Algorithms.

second generation. This decreased demand balances the increased demand for newspace memory caused by a larger newspace threshold. Sometimes the effect of the decreased promotion rate is dramatic, as for the copying collectors with thresholds of 125 and 250 kilobytes. The 250-kilobyte threshold significantly reduces the promotion rate and the memory demand. Above 250 kilobytes, increasing thresholds tend to cause increased memory demand.

This figure also shows more clearly the relative memory requirements of copying and non-copying collectors. Typically, the mark-and-sweep collector uses 30–40% less memory to achieve the same page fault rate as stop-and-copy collection. Incremental and stop-and-copy collection have similar memory demands, differing by less than 10% throughout. The overall memory demand of the programs examined ranges from 4–8 megabytes depending on the algorithm and parameters.

In summary, these page fault rate measurements indicate that the page locality of stop-and-copy collection and incremental collection for reasonable memory sizes is similar. Furthermore, mark-and-sweep collection shows significantly better main-memory locality, showing lower fault rates and typically requiring 30–40% less memory than copying algorithms.

While the evidence from these test programs suggests that compaction associated with copying collection does not significantly improve reference locality, preliminary evidence from another source suggests otherwise. Doug Johnson has informed me that copying compaction in older generations appears to improve reference locality in a large CAD application [48]. In his application, Johnson noted that almost all promoted data remains alive for a large part of the program lifespan (tens of hours). Using Courts' algorithm, which uses incremental collection and delays copying, thus letting the reference pattern determine what is copied, Johnson found that a program with a 200-megabyte working set could execute in a 24-megabyte physical memory with relatively low paging overhead (approximately 7%). These preliminary results suggest that Court's copy-on-demand technique may significantly improve reference locality in older generations for some large applications. Further experimentation is required to determine the extent of the effect. This effect, not measured in my simulations, which are too short to generate large quantities of promoted data, may reduce the relative locality benefits of non-copying collection algorithms.

6.5 CPU Cache Locality

The effect of garbage collection on cache locality differs significantly from its effect on main-memory locality because caches are managed very differently from main memories. First, the replacement unit is much smaller. I investigate 32-byte cache blocks—sixteen times smaller than the 4096-byte virtual memory pages. Second, a direct-mapped cache has a different replacement policy from an LRU-managed main memory. In a direct-mapped cache, block replacement is based entirely on the address referenced. No effort is made to determine which block to replace because only one qualifies for replacement. In such a cache, collision avoidance becomes important because two frequently-accessed addresses that map to the same cache block can cause many cache misses.

Caches are also typically much smaller than main memories, although cache sizes have increased significantly in recent years. Typical cache sizes are now 64 to 128 kilobytes and they will grow as memory chips become more dense. Their size limits the possibility of newspace fitting entirely in the cache, but does not preclude it. Finally, cache latency on a miss is usually measured in tens of cycles, so a cache miss incurs a much smaller penalty than a page fault. Acceptable cache miss rates are generally from 1–5%, depending on possible system bottlenecks like the memory bus.

As we have seen, instruction fetches account for about 85% of the total memory requests in a program. These simulations assume that the instruction and data caches are separated, so collisions between instruction and data fetches do not arise. The high locality of instruction fetches tends to ameliorate the relative poor data cache locality for garbage-collected programs. This section only examines the miss rate in the data cache. The total miss rate is considered in Chapter 8.

In the past, caches have been too small for garbage collections algorithms to have much influence on cache performance. My results show that when the cache is much smaller than the threshold size, threshold size has little effect on miss rate. The combination of generation garbage collection and large caches has introduced the possibility that garbage collection algorithms can affect cache performance. Very recently, Peng and Sohi have considered possible modifications to the cache to enhance cache performance in a garbage collected heap [68]. They conclude in their analysis that an allocate cache operation (mentioned earlier) and a biased-LRU replacement policy for an associative cache will strongly enhance cache performance and lower bus utilization. Biased-LRU replacement identifies and replaces cache blocks that are known to contain garbage. This policy requires that garbage in the cache is detected soon after it becomes garbage. They suggest that reference counting algorithms have this property and should be preferred if the cache provides the recommended hardware support.

While Peng and Sohi's cache enhancements would improve performance, their analysis is limited in several ways. First, the programs they measure are small benchmarks, mostly from the Gabriel suite, that perform on the order of 20,000 to 200,000 heap references. They conclude that a fully-associative cache of 3640 words should be large enough to capture the locality of the programs. The cache sizes they investigate range from 512 bytes to 64 kilobytes. Finally, they fail to investigate generation collection algorithms that provide a significant enhancement to locality. My results, which focus on the effect of garbage collection algorithms in a relatively simple cache, differ significantly from Peng and Sohi's. In particular, the long reference strings from the programs I measure (20,000,000 references) require larger caches to capture the program locality, even with generation garbage collection. The major reason for the difference is that non-benchmark programs use large long-lived data structures and reference system objects with higher frequency than small benchmark programs. These objects behave much differently than the short-lived objects measured by Peng and Sohi.

6.5.1 Direct-Mapped Caches

Figure 6.5 shows the data cache miss rates for different cache sizes and threshold sizes with the stop-and-copy algorithm. For caches that do not fully contain newspace, we see a small improvement (roughly 1%) by doubling the cache size. When the cache does contain newspace, however, the fit improves dramatically. For a two megabyte cache, the effect of threshold size on miss rate is pronounced, suggesting that for sufficiently large caches, generation garbage collection can be used to improve data cache locality to the point of almost entirely eliminating cache misses. This result is significant because in systems of the future, as CPU speeds continue to increase, memory accesses may become the major bottleneck to system performance. I conclude that generation garbage collection can be used to effectively tradeoff CPU speed with memory system capacity to achieve balanced performance.

The figure also shows the large difference between the performance of a one megabyte cache and a two megabyte cache, even when they both fully contain newspace. The difference in miss rate is probably caused by a failure of the one megabyte cache to contain many objects frequently referenced in oldspace. Even with newspace fully contained, oldspace references play a major role in cache performance.

Figure 6.6 shows the cache miss rates for the mark-and-sweep algorithm. Overall, the cache sizes needed to obtain the same miss rate as the stop-and-copy algorithm are smaller. As expected, using a non-copying algorithm provides approximately the same miss rate as a copying algorithm in a cache of half the size. Since cache memory chips are expensive, this result suggests that non-copying algorithms are significantly more economical than copying algorithms in the presence of generation collection. Interestingly, however, for large caches and small threshold sizes, the stop-and-copy algorithm performs better than mark-and-sweep, again because the mark-and-sweep algorithm is promoting more objects that decrease cache locality.

Another difference between mark-and-sweep and stop-and-copy collection is the difference in miss rates for one and two megabyte caches. While the stop-and-copy algorithm shows a big difference, there is hardly any difference in the miss rate for these cache sizes with mark-and-sweep collection. One explanation for this similarity is that mark-and-sweep collection places objects of the same type on a page. In particular, symbols in oldspace have much better spatial locality in the mark-and-sweep algorithm. In the stop-and-copy algorithm, symbols are distributed evenly throughout oldspace. Because many oldspace references are made to to symbols (particularly for special variable binding), these references will be less likely to miss in a direct-mapped cache if oldspace symbols are grouped together.

Finally, Figure 6.7 compares the cache miss rates for the three algorithms for several cache sizes. As with page fault rates, the miss rate of the incremental algorithm is very close to that of the stop-and-copy algorithm. Non-copying collection clearly offers the best cache miss rate. These results suggest that the effect of garbage collection algorithm and threshold size is pronounced enough that the CPU cost versus locality tradeoff should be evaluated in

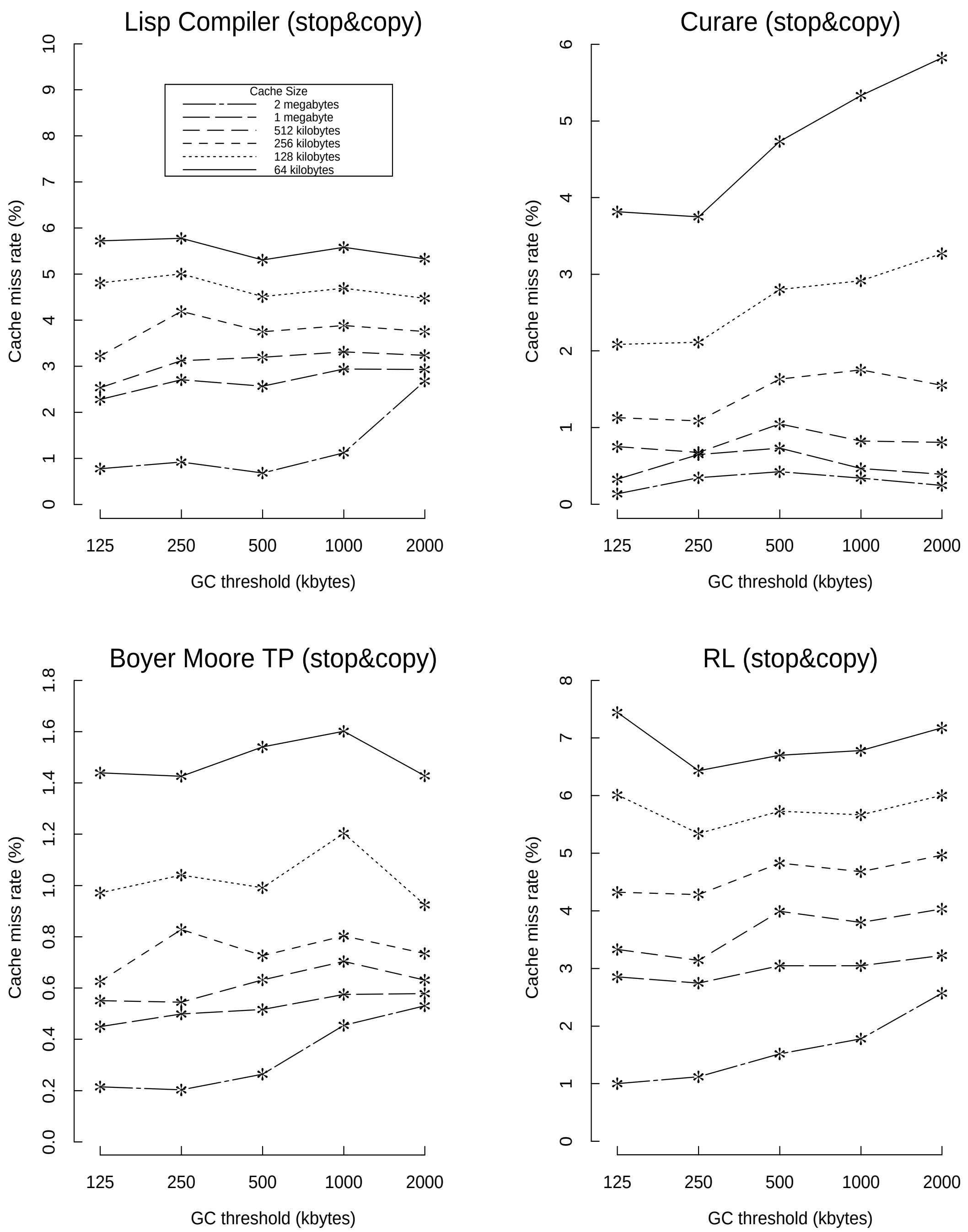

Figure 6.5: Cache Miss Rates for Stop-and-Copy Collection.

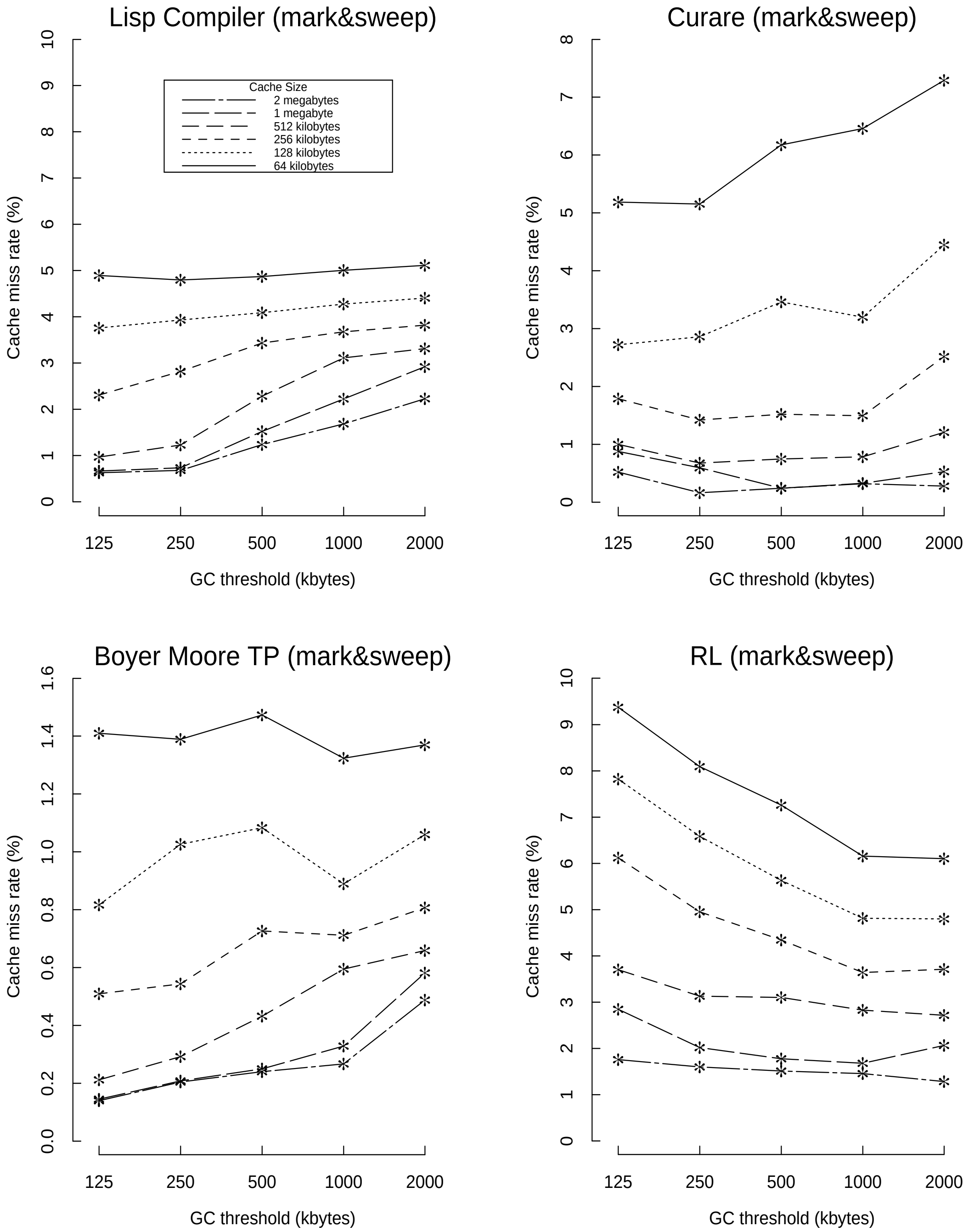

Figure 6.6: Cache Miss Rates for Mark-and-Sweep Collection.

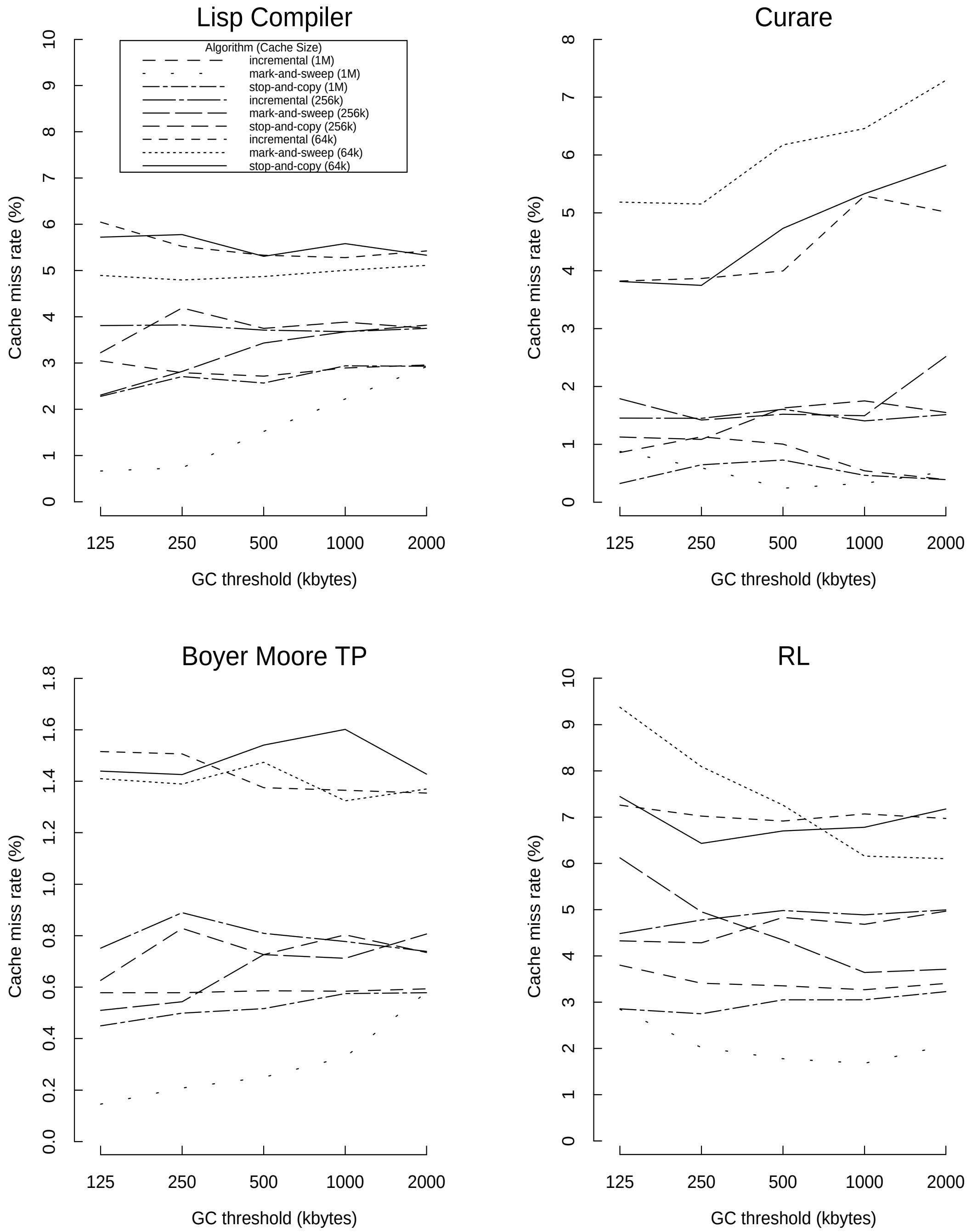

Figure 6.7: Cache Miss Rates for Three Collection Algorithms.

systems where cache performance is important. In Section 8.3.2, the effect of cache locality on total performance for both uniprocessors and multiprocessors is discussed. For now, we can conclude that mark-and-sweep collection offers clear advantages over copying collection algorithms for most cache, memory, and threshold sizes.

Chapter 7

Availability

This chapter investigates the effect that garbage collection algorithms have on the interactive availability of programs. In particular, because incremental collection avoids perceivable pauses, this chapter focuses on the pauses caused by the stop-and-copy and mark-and-sweep algorithms discussed in Chapter 4. The effect that threshold sizes and promotion policies have on pause length, promotion rate, and collection overhead are considered. While large threshold sizes always result in lower CPU overhead when enough memory is available, the longer pauses associated with larger thresholds may be unacceptable. This chapter quantifies the relationship between threshold size and pause length.

Collection pauses are a problem because I am assuming a model of use where a programmer starts a CPU (and allocation) intensive job (e.g., a compile) running in the background, and then uses the same Lisp system for more interactive tasks such as editing or reading mail. Because garbage collection of one Lisp process will interfere with all processes that share the same address space, the interactive programs will be delayed by garbage collection, disrupting the user. This model, where Lisp processes share an address space, is a common one for Lisp workstations.

7.1 Pause Histories

Before examining the relationship between pause length and collection parameters, I present data illustrating the observed pauses for three of the four test programs. Figure 7.1 shows the frequency and duration of collection pauses for the stop-and-copy and mark-and-sweep algorithms with a *pause history* diagram.

The patterns of collection illustrate the basic problems of pause length and promotion in a generation garbage collector. Each plot shows the collection patterns for three different threshold sizes. The smallest threshold results in more frequent, shorter collections. However the smallest threshold also results in more rapid promotion of data, and so collections of the second generation (possibly lasting several seconds) occur more frequently. While

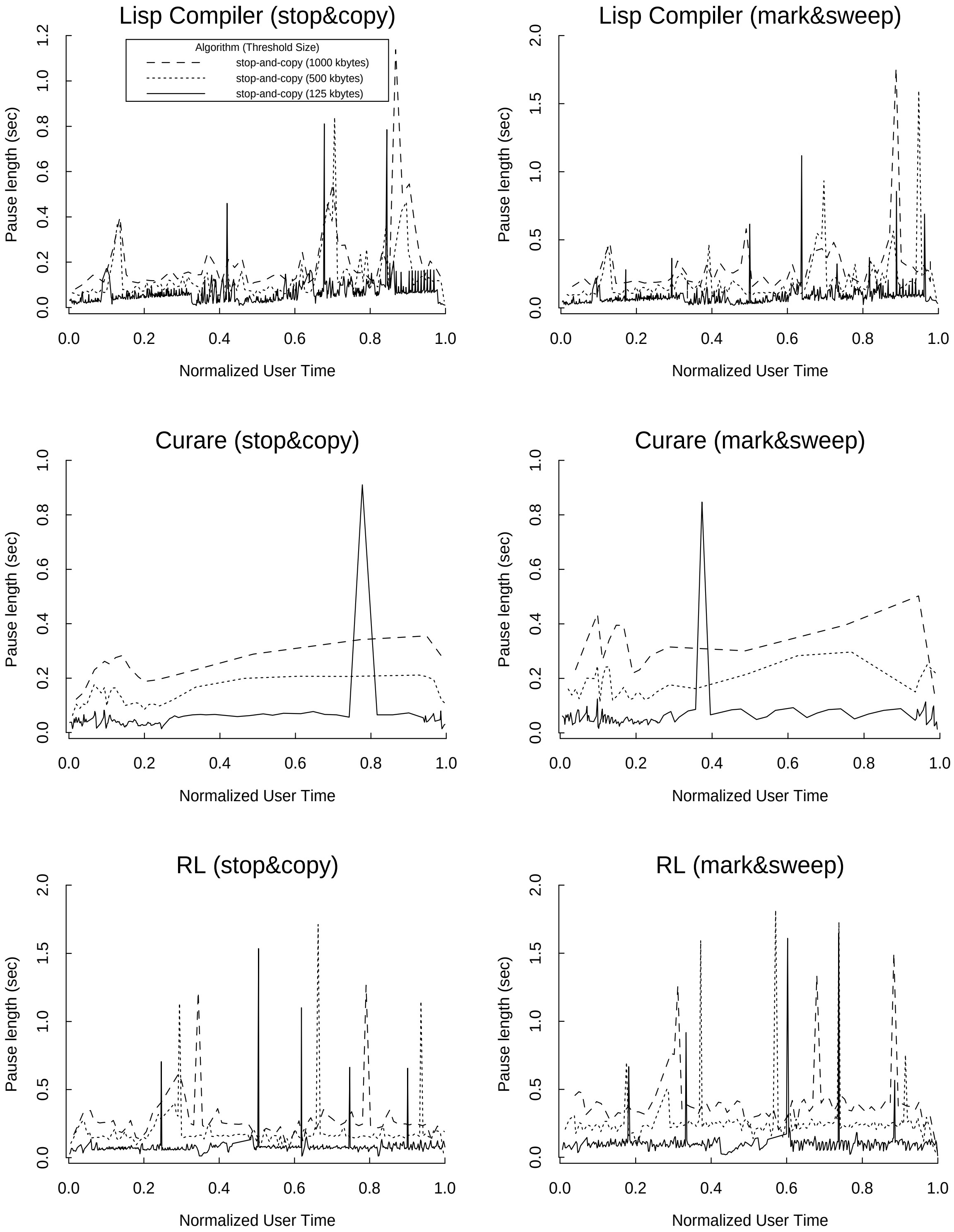

Figure 7.1: GC Pause Lengths as Function of Time.

average pause length is decreased with small thresholds, the frequency of noticeable second generation pauses may actually increase with small thresholds unless care is taken to avoid promotion.

In these pause histories, the promotion policy used by the stop-and-copy algorithm is *copy count* promotion, where objects are promoted after they have been copied three times during garbage collection. For the mark-and-sweep algorithm, the policy is to promote the entire generation after three collections (*en masse* promotion). The figure illustrates that second generation collections are more frequent for mark-and-sweep collection due to increased promotion from the en masse promotion policy. Another effect of en masse promotion is that collection times are less uniform, as illustrated best by Curare. With en masse promotion, immediately after a promotion, newspace is empty and less data is marked when the threshold is reached. When newspace is promoted, collection takes longer because copying requires more time than just marking. This non-uniformity leads to the slightly more spikey appearance of the mark-and-sweep pause history diagrams.

While the pause histories give us an idea of the overall patterns of collection, they do not provide an exact understanding of the performance tradeoffs available over a range of threshold sizes and promotion policies. The easiest way to explore this design space is to use object lifespan distributions.

7.2 Object Lifespans

The distribution of object lifespans largely determines the duration and frequency of garbage collection pauses given a particular promotion policy and threshold size. Using MARS, I have measured the lifespans of program objects in units of program references, a fine-grained unit of time. The lifespan of an object is defined to be the number of program references between the first and last reference to the object, even though the object may remain reachable for some time after the last reference is made. The object is assumed to be garbage immediately after the last reference. Alternate assumptions, such as assuming the object is dead only when it is actually reclaimed, are not fine-grained enough, as collections are relatively infrequent. To determine the exact lifespan by traversing all objects and noting the live ones after every reference would be prohibitively expensive to compute.

Figure 7.2 shows the survival rates (i.e., the fraction remaining alive beyond a particular lifespan) for different object types in the four test programs. In addition to survival rates for classes of types, the total survival rate for all objects is shown. Predictably, the total is close to the cons curve because so many cons cells are allocated. Although the overall shapes of the plots differ significantly, several trends are common to all programs. First, in all cases the survival curves are quite steep. After one second less than 20% of all objects remain alive in most cases—after ten seconds less than 5% of the objects remain alive. Second, the curves decrease very steeply initially and then flatten out. Very short-lived objects are probably temporary objects allocated only for the duration of an expression or function call and then discarded. A second sharp decrease in the survival curve occurs after

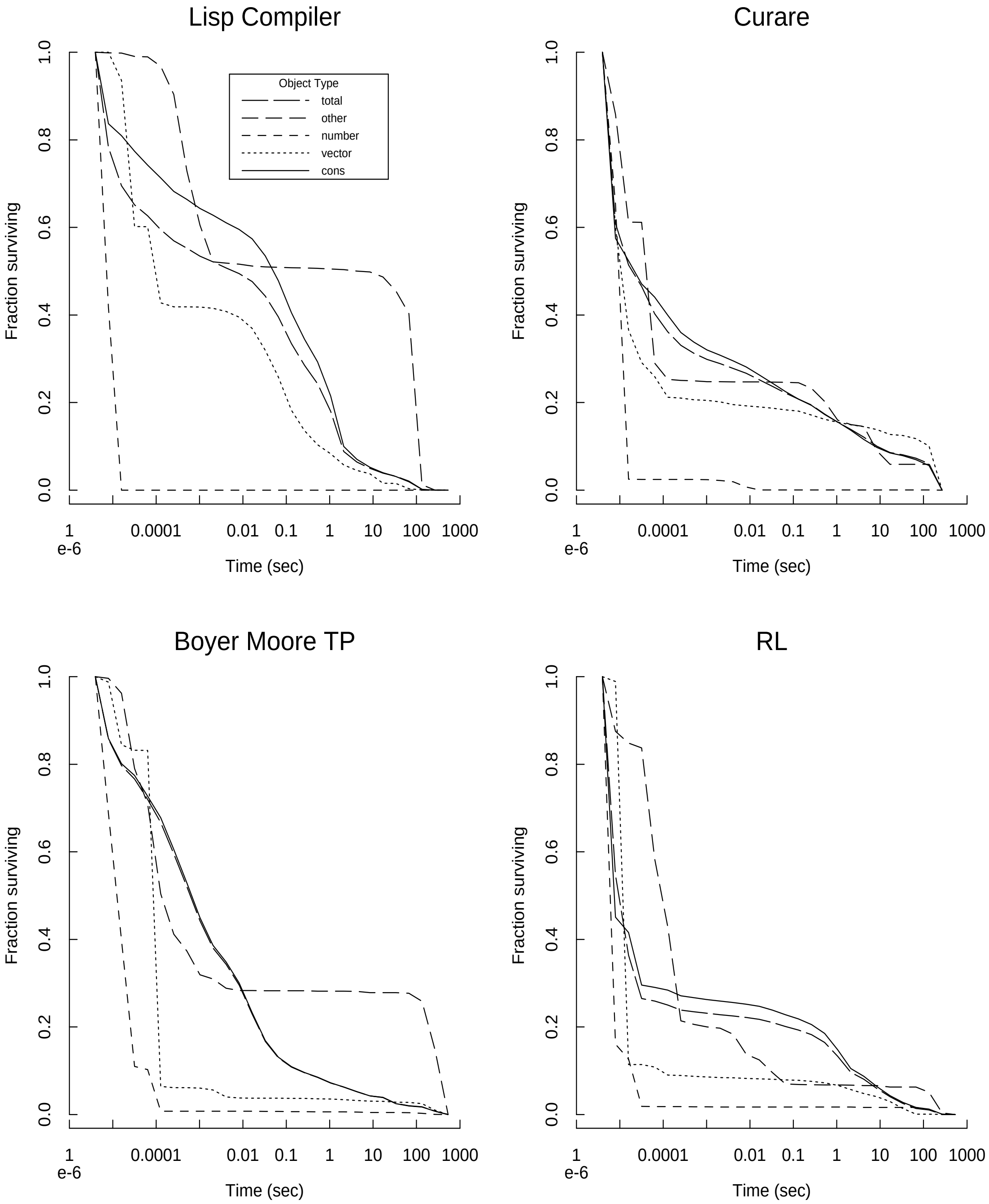

Figure 7.2: Survival Distribution of Objects Referenced by Object Type.

about one second in the compiler and RL programs. This decrease represents the deaths of objects allocated for the duration of a program subtask. In the compiler, the subtask is the compilation of a single top-level form. In RL, the subtask is instruction scheduling for a basic block.

Interestingly, in both cases the subtask takes a few seconds or less to compute. I surmise that subtasks that last a second or less are commonly found in systems where development takes place interactively. If the basic subtask took longer to compute, then the pause required for that computation would interfere with the interactive flow of program development. I investigate the consequences of this possibility in Chapter 8, which considers how object lifespans may change when processor performance increases.

There are other interesting trends in Figure 7.2. The survival curves of different types of objects are quite different. For example, numeric objects invariably have very short lifespans and clearly represent temporary values generated as intermediates in arithmetic expressions. Some curves also show rapid decreases at specific lifespans (in particular, objects with type "other"). There are two reasons for these very sharp changes in survival rate. The first reason is statistical. Classes of types of which few objects are allocated (e.g., "other" objects) will show more variation in survival rate than classes where many objects are allocated (e.g., cons objects) because the sample size is smaller. Another reason for sharp decreases in the survival rate is that some objects are allocated for very specific lifespans. These are objects allocated for a task with a fairly constant duration, like the execution of an I/O operation or the evaluation of any function whose execution time is independent of the value of its arguments.

7.3 Discrete Interval Simulation

DIS (Discrete Interval Simulator) is a program that I wrote to augment MARS. DIS takes object lifespan and allocation information from a particular program as input and simulates the behavior of garbage collection algorithms at a very high level. DIS works by breaking up the program execution into discrete intervals. One input to DIS is a sequence of the number of bytes the program allocated in each interval. A second input to DIS is the lifespan distribution of the program's objects. By knowing the number of bytes allocated in any interval i, with lifespan information, DIS can predict how many of those bytes are still in active use at any later interval j.

For each simulated interval, DIS maintains a count of the current number of active bytes in all previous intervals. When an allocation threshold is reached, DIS simulates a garbage collection. DIS maintains information about each time interval in addition to the number of active bytes. The number of times the bytes have been copied, as well as the generation in which they are currently stored is noted. When a new interval is simulated (call it interval i), the values associated with the interval are initialized. The active byte count is set to the number of bytes allocated, the generation those bytes occupy is newspace, and the copy count is set to zero (the bytes have never been copied). As later intervals are simulated, interval i ages, and the number of active bytes it contains decreases. Eventually, a newspace garbage collection is simulated. Intervals stored in newspace are modified to reflect that they have been copied (the copy number is incremented) or promoted (the generation number is incremented). By recording how many active bytes are transported or promoted, an estimate of the cost of the collection is possible.

Using this method, the frequency and duration of collections of all generations can be predicted if the collection duration is assumed to be proportional to the number of bytes collected. Furthermore, the promotion rate of objects to older generations can be measured. DIS does not provide any information that is not also provided by MARS. Its advantage over MARS is that the collection performance of an algorithm under a set of parameters can be obtained in a matter of seconds instead of the hours required to obtain the information using MARS. Therefore, it can be used to investigate garbage collection behavior for a much longer period of time than is possible using MARS. In particular, the pause length and frequency of collections of older generations can be measured.

The major drawback of DIS is that by assuming a uniform lifespan distribution and breaking program execution into fairly large discrete intervals, subtleties of the actual program are lost. For example, object lifespan may strongly depend on time—long-lived objects may be allocated mostly at the start of a program. Lifespan may vary with allocation rate—objects allocated during periods of rapid allocation may have shorter lifespans. While DIS misses such dependencies, I will show that its predictions of performance are often close to those measured using MARS.

The remainder of this chapter uses DIS to examine two things: how promotion rate, pause length, and CPU overhead vary with promotion policy and threshold size in newspace, and how promotion rates influence the frequency and duration of collections in older generations.

7.4 Newspace Collection

While the main focus of this chapter is processor availability and garbage collection pauses, in this section we will see that overhead, locality, and pause length are all closely related. Using results from DIS, we will first examine pause length, then CPU overhead, and finally promotion rates caused by collections of newspace. Throughout this chapter, performance is considered to be a function of threshold size and promotion policy. The promotion policy is further divided into the promotion strategy (copy count or en masse promotion) and the number of copies (or flips) performed before promotion takes place (the *copy number* parameter). In these results, the stop-and-copy algorithm uses copy count promotion and the mark-and-sweep algorithm uses en masse promotion.

One of the goals of this evaluation is to determine if en masse promotion is an effective promotion strategy as compared to copy count promotion. If en masse promotion cannot offer competitive pause lengths and promotion rates when compared with copy count promotion, then the mark-and-sweep algorithm proposed may not be acceptable. Another goal of the evaluation is to examine the space of design parameters to determine what parameter values avoid noticeable collection pauses and offer low overheads.

7.4.1 Pause Lengths

Figure 7.3 shows the predicted pause lengths for three applications and two collection algorithms where the copy number was varied from zero to ten over the range of threshold sizes. In addition, the value obtained using MARS, where the copy number was set to three, is provided to check the validity of the DIS results.

When objects are promoted immediately (i.e., the copy number is zero) there is no difference between copy count promotion and en masse promotion. Immediate promotion yields the shortest pauses because fewer objects are copied back and forth in newspace. While what is "acceptable" for a pause length is subjective, pauses longer than half a second are definitely noticeable to an interactive user and probably disruptive. Since newspace collections are relatively frequent (all of the configurations shown collect more often than once per minute), non-disruptive pauses are essential. So, while thresholds larger than two megabytes may be attractive due to low CPU overhead (assuming enough physical memory is available), they also carry the disadvantage of long pauses (one second or more). In Figure 7.3 we see that even two megabyte thresholds cause noticeable pauses unless objects are almost immediately promoted. At 500 kilobytes, pauses are typically half as long as for the two megabyte threshold and for copy counts of three or less the pauses are short enough to go unnoticed (0.2–0.3 seconds).

Comparing the mark-and-sweep plots with the stop-and-copy plots, we see the trends are similar but mark-and-sweep durations are only about 80% of the stop-and-copy durations given similar threshold sizes and copy numbers. As we noted in Section 7.1, en masse promotion results in a greater variation in pause lengths, with a slightly lower average due to the increased promotion. The *average* duration, as shown in the figure, may also be somewhat misleading. Since collections that promote take longer than collections that mark, and because collections that promote are relatively frequent, even though mark-and-sweep has a lower average duration, the longer pauses experienced will be as long as the average stop-and-copy pauses, and mark-and-sweep will have no advantage.

Finally, the MARS results indicate that, while the DIS curves are probably more uniform than curves measured with MARS, they are not incorrect by more than 20% in most cases.

7.4.2 CPU Overhead

Having established that very large thresholds may cause unacceptable pauses, we now examine the relative CPU overheads for the same range of parameters. The overheads are calculated as a function of bytes copied (or marked), assuming the collection cost is proportional to the amount copied. For these overheads, the costs of maintaining the write barrier are not considered. As the unit of comparison, I chose copy count promotion with a copy

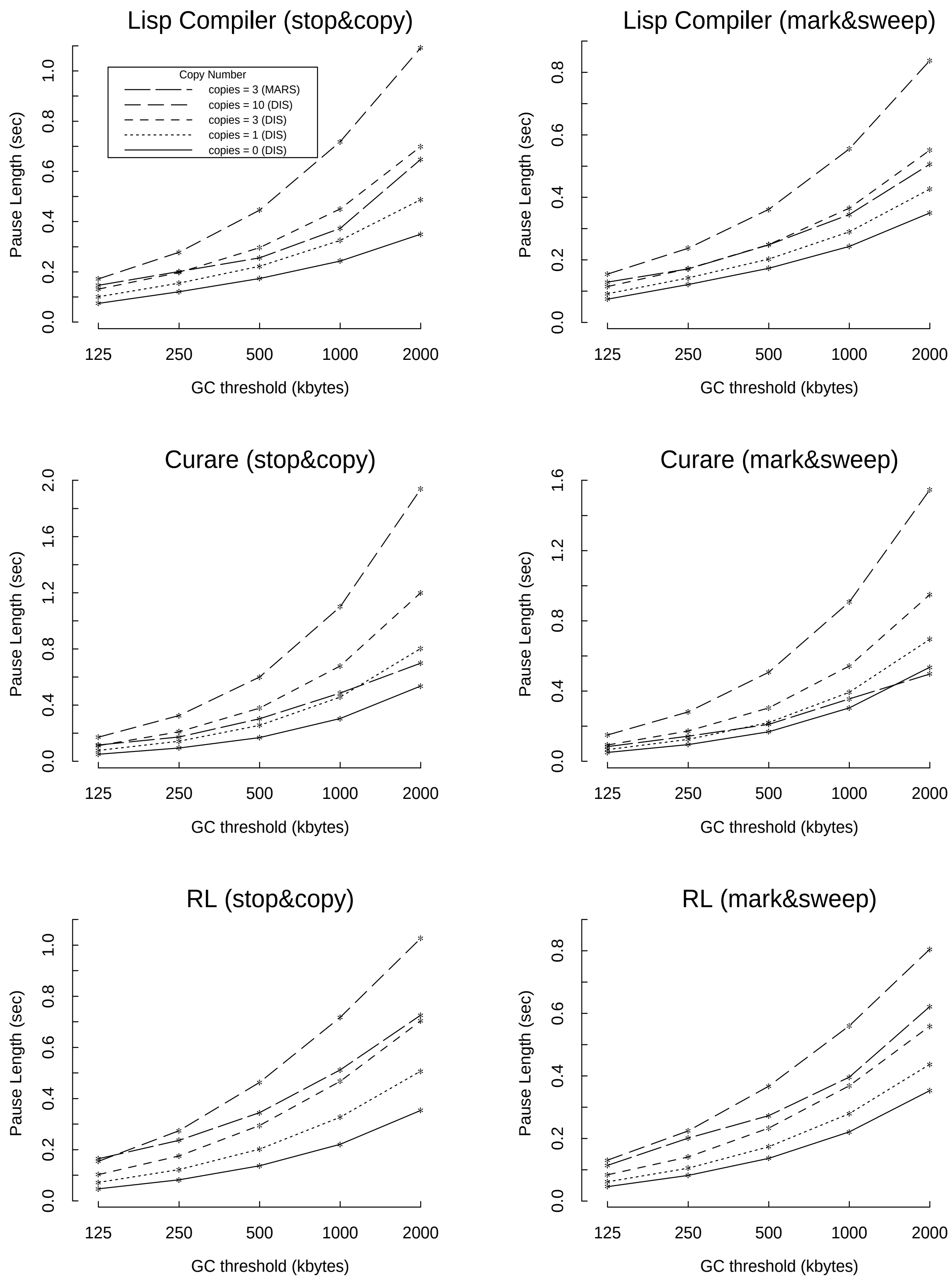

Figure 7.3: Pause Lengths for Three Applications

number of three and a threshold size of 500 kilobytes because this combination appeared to provide reasonable pause lengths (and low promotion rate, as we will see).

Figure 7.4 shows the relative overheads. For reference, each unit of overhead corresponds to approximately 15–20% of total execution time, depending on the application. As with pause lengths, immediate promotion results in less CPU overhead, sometimes more than a factor of five less than promotion after ten collections. Even with small threshold sizes, immediate promotion results in less overhead than the baseline configuration. The drawback of immediate promotion is the high promotion rate, as we will see in the next section.

Looking at the CPU overhead, we see the cost being paid to avoid long pauses. If threshold sizes are increased to two megabytes, overhead decreases to half the baseline cost. Incremental collection offers the advantage of allowing large thresholds without the long pauses, thus increasing the efficiency of collection. Unfortunately, for the threshold sizes considered, the added overhead does not make up for the added cost of implementing the read barrier with stock hardware. Still, with threshold sizes larger than two megabytes, incremental collection may be very competitive with stop-and-copy collection, even on stock hardware. Non-incremental collection carries the hidden cost of needing to size thresholds to avoid long pauses. For these programs, that cost is about 10% CPU overhead with the additional cost of extra promotion.

The figure also shows the CPU costs of increasing the copy number. Collecting objects three times before promotion can more than double the CPU overhead when compared with promoting objects immediately. The drawbacks of the increased promotion rate must be evaluated carefully before the alternative of immediate promotion is discarded.

Comparing copy count promotion with en masse promotion, we see that en masse promotion results in lower overheads. This is true for the same reason that a low copy number produces lower overheads, because when objects are more rapidly promoted, they are no longer copied within a generation as much and collections take less time. So far, we have seen that en masse promotion and promotions with low copy numbers result in shorter collection pauses and significantly less CPU overhead, both favorable conditions. The next section examines the increased promotion rate caused by these approaches.

7.4.3 Promotion Rates

We now look at the promotion rates of the algorithms over the range of parameter values. Figure 7.5 shows that the promotion rates vary widely in the design space, ranging from less than 1% to 30% of all bytes allocated. The figure immediately shows the benefits of allowing one copy before promotion for all threshold sizes (i.e., copy number = 1). The relative benefit of increasing the copy number by two is less significant, but the relative significance increases as threshold size increases. For large thresholds, the fraction of objects that survive three copies is quite small. Increasing the copy number beyond three does not appear to be cost-effective in preventing further promotion.

The figure clearly shows that promotion rates of 2–5% are almost unavoidable for the

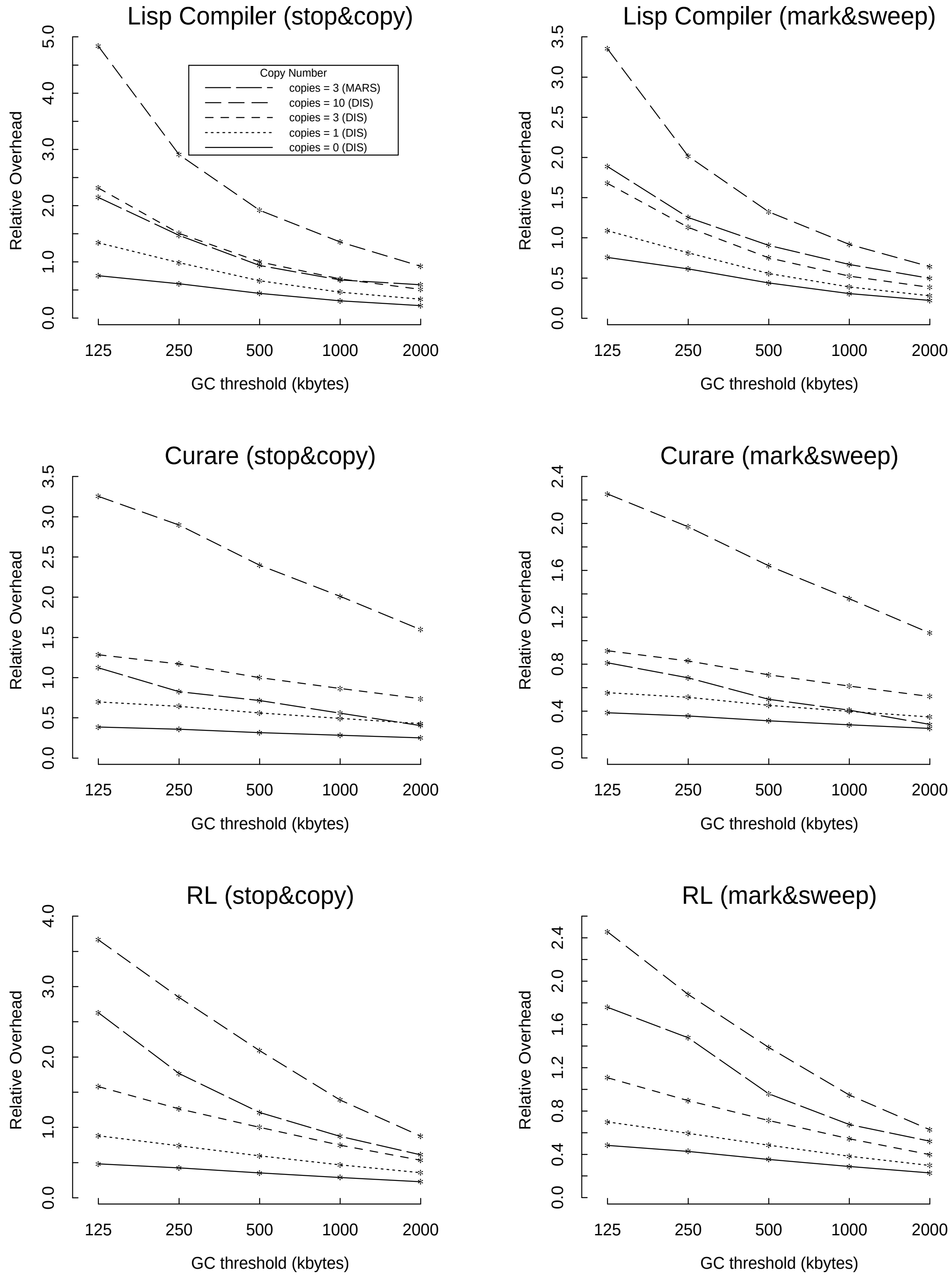

Figure 7.4: Relative CPU Overhead for Three Applications

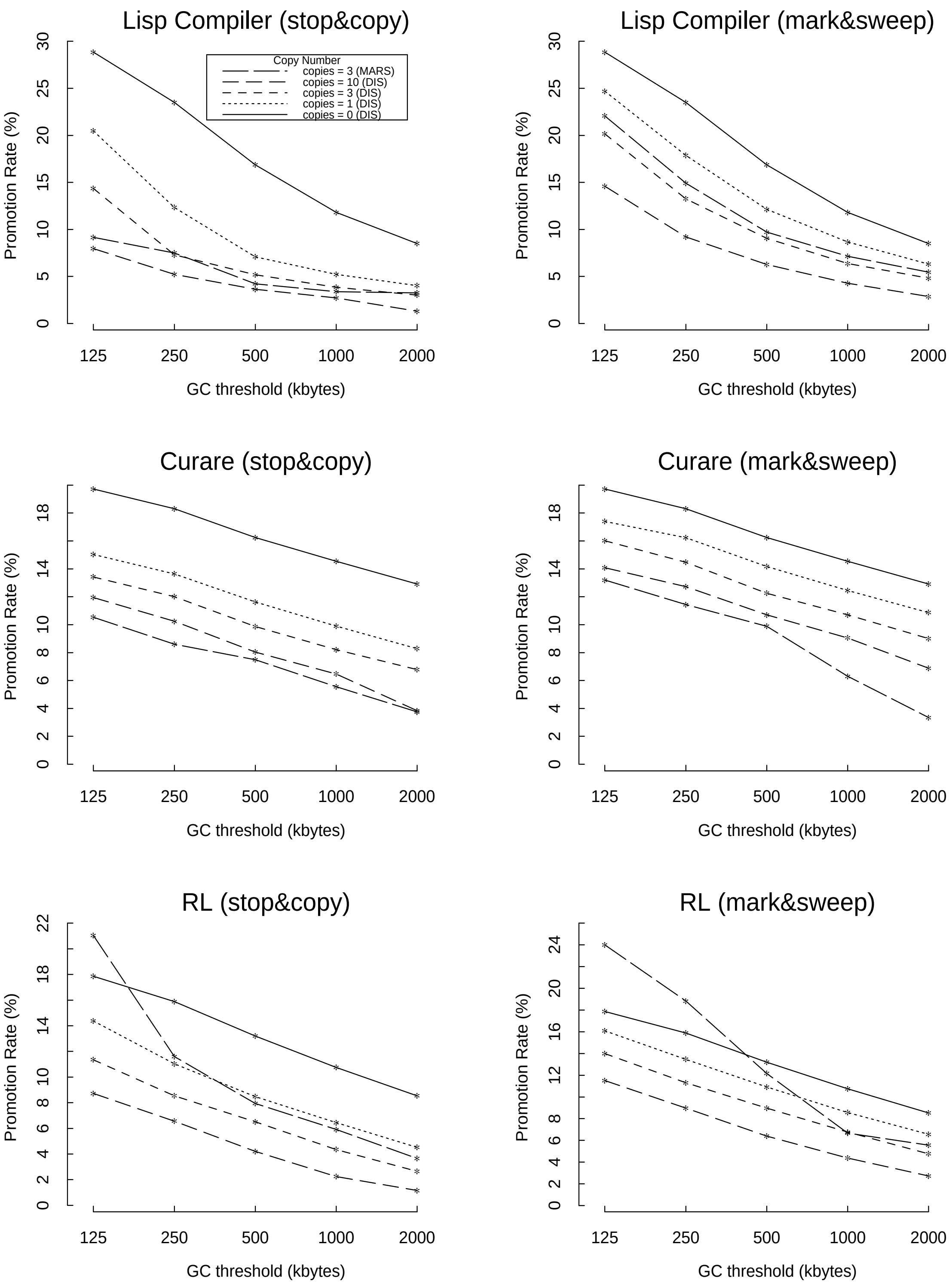

Figure 7.5: Promotion Rates for Three Applications

threshold sizes examined. If thresholds of 500 kilobytes or less are used to avoid noticeable pauses, promotion rates of 4–10% appear likely.

En masse promotion results in an increased promotion rate, promoting more that 20% more data than copy count promotion with small thresholds, and up to 100% more data with larger thresholds. Larger thresholds show a relatively increased promotion rate for en masse promotion because en masse promotion always promotes some very young objects, whereas copy count promotion can completely avoid promoting any younger objects. As threshold sizes increase, copy count promotion becomes relatively more efficient because fewer objects survive. A similar argument can be made comparing en masse promotion and copy count promotion over a range of copy numbers. As we see, setting the copy number to zero makes the two strategies behave identically. As the copy number increases, the advantage copy count promotion has over en masse promotion also increases.

For a 500 kilobyte threshold with a copy number of three, en mass promotion promotes 50–80% more data than copy count promotion. One way to evaluate this difference is to examine the tradeoffs that must be made to reduce the en masse promotion rate down to the copy count level. The two ways to reduce the promotion rate are to increase the threshold size (and hence memory usage and pause length), or increase the copy number parameter beyond three. Leaving the copy number at three would require more than doubling the threshold size to achieve the same promotion rate. Leaving the threshold size the same, the copy number would have to be increased to more than ten to sufficiently lower the promotion rate. A copy number of ten results in a CPU overhead more than three times higher than that of a copy count strategy with the same promotion rate.

En masse promotion incurs a significantly higher promotion rate that either requires more memory or more CPU cycles to reduce the rates to those of copy count promotion. The next section examines the long term effects of a higher promotion rate on system performance.

7.5 Collection of Older Generations

Because collections of the second generation are relatively infrequent, their CPU overhead is negligible compared to the overhead of first generation collections. The problem with second generation collections is that because promoted data tends to be relatively long-lived, much of the data in the generation requires collection, resulting in potentially disruptive pauses.

To decrease the frequency of collection, the second generation is typically sized larger than the first generation. By giving second generation objects more time to die, larger sizes in the second generation can actually result in shorter collection pauses, as we shall see. However, a larger second generation typically takes longer to collect and in addition will cause greater disruption to the reference locality of the program, especially if the second generation is too large to fit comfortably in the physical memory.

Because the test programs only execute for several hundred seconds, to gather data for longer periods of time, the DIS simulation executed each test program as many times as was necessary to cause at least 100 garbage collections of the second generation. In general we expect k iterations of a problem of size N to behave differently from one iteration of a problem of size kN. By performing k iterations, the assumed behavior resembles the way a developer might use a Lisp machine—repeatedly compiling or simulating the same file or circuit. Chapter 8 discusses the consequences of longer running programs. In all cases, the copy number used in newspace and the copy number in the second generation are assumed to be three.

In the experiments conducted using DIS, generation sizes ranging from half a megabyte to eight megabytes were examined. For each threshold size, four promotion rates from newspace were considered: 2, 4, 10, and 20%. The 2% rate corresponds to a lower bound better than almost all the rates predicted by DIS in the previous section. A 5% promotion rate is typical for copy count promotion with a newspace threshold size of one megabyte. A 10% promotion rate is typical for en masse promotion with the same parameters. Finally, a 20% promotion rate corresponds to a typical rate for a one megabyte threshold with immediate promotion (copy number = 0). I include this rate to determine if such an approach (which shows significantly short pauses and overhead) is viable. We will look at predicted collection frequencies and pause lengths. In all cases, promotion rates to the third generation are negligible.

Figure 7.6 shows the predicted collection frequencies as a function of promotion rate and threshold size. With a promotion rate of 20%, collections occur every several minutes, even with an eight megabyte threshold. Programs with a 20% promotion rate collected almost 20 times more frequently than programs with a 2% promotion rate because the lower promotion rate gives more objects time to die between collections and makes collections more efficient. Between the 5% and 10% promotion rates, the 10% rate causes collections about two and a half times as frequently. For thresholds under eight megabytes, a 10% rate often requires collections every several minutes or less. If these collections cause noticeable pauses, they are probably unacceptable.

Increasing threshold size has a better-than-linear effect in decreasing collection frequency. But even so, unless the promotion rate is very low, second generation collections occur at least once per hour, even with eight megabyte thresholds. If the large thresholds cause significant paging, even when the the actual CPU required for collection is small, users may experience disruptive system behavior.

Given that promotion rates of 5% and more cause collections every fifteen minutes or so, the duration of these collections is very important. Figure 7.7 shows the predicted pause lengths for second generation collections. For 2% newspace promotion rates, the predicted pause length appears to be relatively insensitive to threshold size and very short, even for large thresholds. This behavior is a consequence of the lifespan data input into DIS. All the programs measured ended after a matter of a few minutes. Object lifespans measured were never longer than the program lifespan. Because the 2% promotion rate causes second generation collections to be infrequent (relative to the program lifespan), no

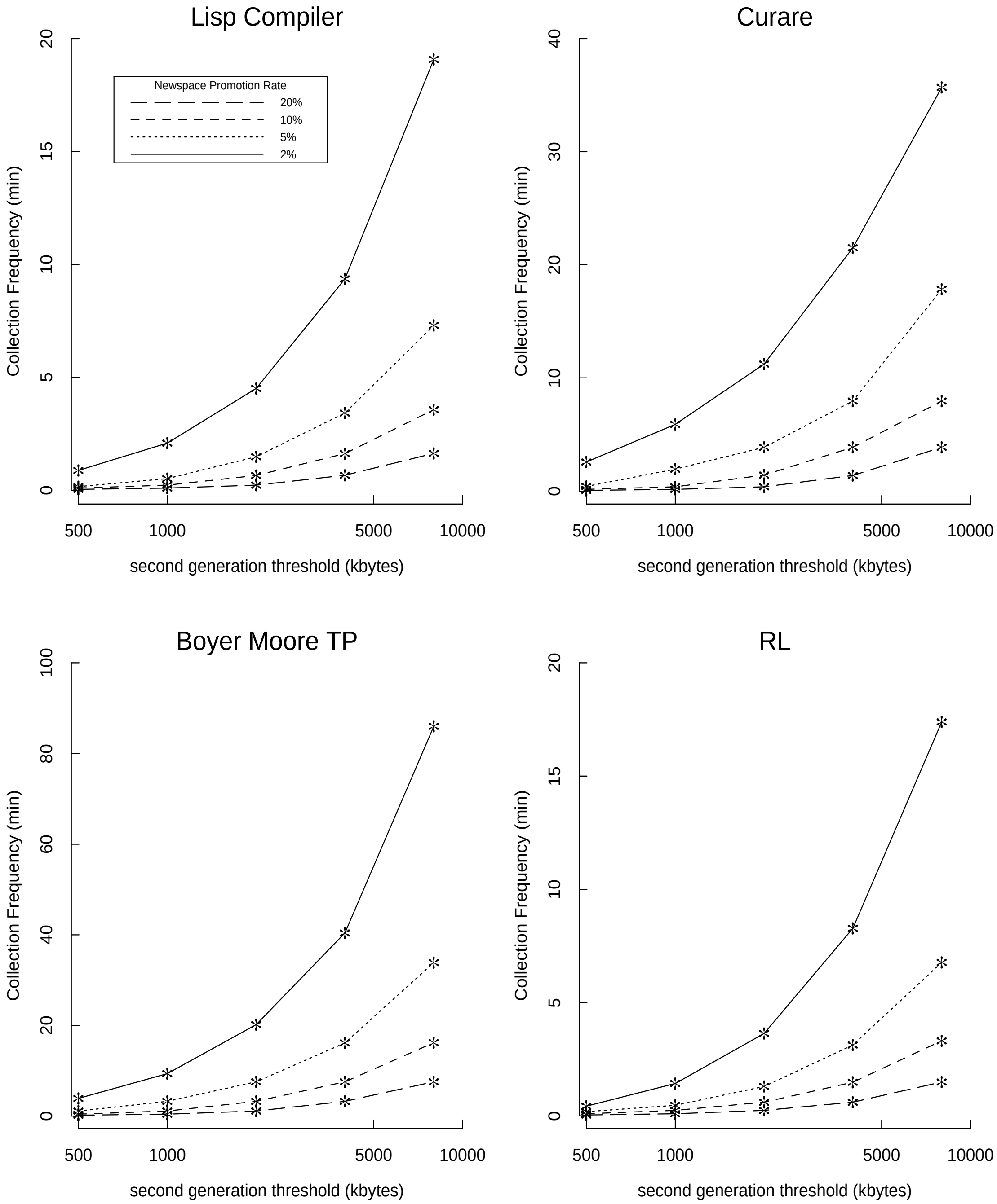

Figure 7.6: Second Generation Collection Frequencies for Four Applications

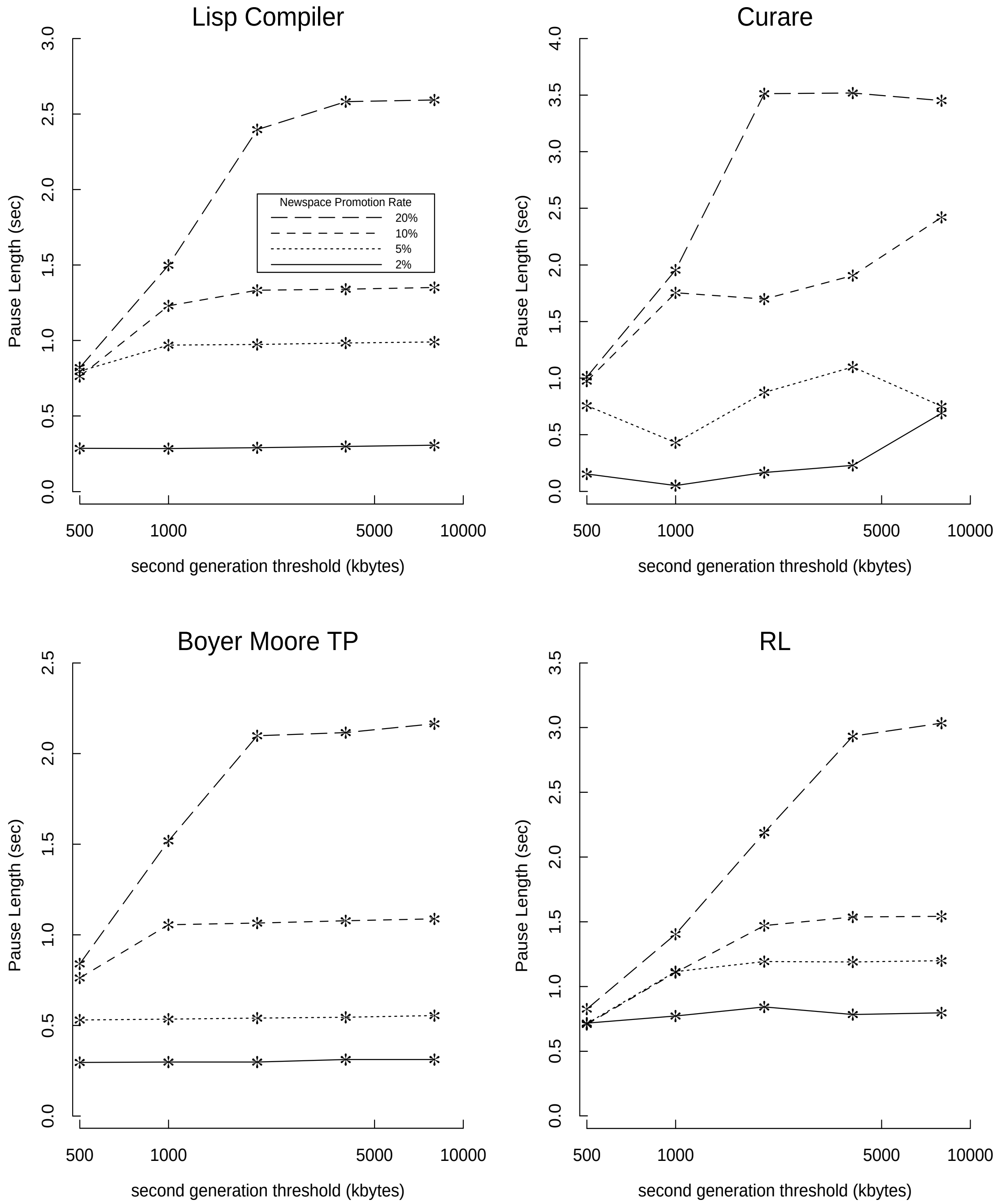

Figure 7.7: Second Generation Pause Lengths for Four Applications

matter what the threshold size, the objects collected are only from the current iteration of the program execution—no very long-lived data accumulates because it was not measured. As a consequence, for very infrequent collections, only a few objects are collected no matter what the threshold size, hence the insensitivity.

This behavior is both a true consequence of running programs and an artifact of my measurements. Most data allocated by a program is shorter-lived than the program itself. However, Lisp systems typically have some program-generated objects that last for hours and days. Because my methods do not measure these objects, I cannot predict the effect they have on overall performance, but I suspect it is limited for a simple reason. Objects lasting beyond a program execution are typically either results that are stored permanently in a file (e.g., simulation results) or are integrated into the environment to become part of the system (e.g., loading an object file). System data cannot grow too quickly or systems would become unacceptably large. Objects stored permanently are then unnecessary and can be discarded by the user to become garbage. The result is that data living across program executions is typically either small or short-lived. If the test programs had taken longer to execute, the results in Figure 7.7 would also be different. The next chapter explores the consequences of programs that take longer to execute.

When second generation collections occur frequently enough that active program data accumulates in the second generation, we can see the negative effect of threshold size on pause length. If the trends are extrapolated (i.e., longer running programs with similar lifespan distributions are considered), pauses of five seconds or more are possible with an eight megabyte threshold. Even the programs measured show multisecond pauses for 10–20% promotion rates and thresholds larger than two megabytes.

For programs with the lifespan distributions measured, the 5% promotion rate shows very definite advantages over the 10% rate. While one second pauses are still noticeable, if they occur every fifteen minutes or so, they do not constitute a serious disruption. Furthermore, because the 5% rate is also relatively insensitive to threshold, larger thresholds may not cause longer pauses, allowing the frequency to be decreased even more (if sufficient physical memory is available).

The figure demonstrates that while immediate promotion (and hence a 20% promotion rate) makes first generation collections more attractive, the second generation collections are very frequent and definitely noticeable. The 10% promotion rate appears to border on unacceptable (e.g., 1–2 second pauses every 5–10 minutes), even by fairly loose standards. By significantly reducing the promotion rate, copy count promotion shows a clear advantage over en masse promotion.

This chapter has shown the effect of threshold size and promotion strategy on collection CPU overhead, duration, and frequency in the first and second generations. Thresholds larger than one megabyte greatly reduce promotion but produce noticeable pauses. Thresholds less than 250 kilobytes result in increased overhead and promotion that interferes with program reference locality, as we saw in Chapter 6. Thresholds near 500 kilobytes result in CPU overhead of approximately 10-20%, half second collection pauses, and promotion rates of 5–10%. Furthermore, en masse promotion results in promotion rates 50–100% higher than copy count promotion, with the relatively higher rate coinciding with larger thresholds. In the next chapter, we see how these performance metrics might change in systems of the future.

Chapter 8

Faster Processors and Multiprocessors

So far, this thesis has investigated various aspects of the performance of collection algorithms executing on workstations using current processor technology. The processor speed assumed is approximately that of a Sun4/280, the processor on which the studies were conducted. This chapter considers the effects of possible future technology on the performance of collection algorithms. As mentioned in Chapter 2, a performance evaluation tool should help designers plan for future technology trends. This chapter uses MARS and DIS for this purpose.

Two obvious technology trends require investigation. The first is a rapid increase in the processor performance available in relatively inexpensive workstations. While the Sun4/280 is rated at approximately eight to ten MIPS, processors with significantly higher performance are already available. Fabrication and semiconductor technology, as well as advances in architecture design pave the way for even faster processors to appear in the years to come. Section 8.1 examines the effects of faster CPU's on the performance metrics we have used to evaluate different collection strategies.

Another technology trend is the availability of commercial multiprocessors. While current implementations of symbolic languages on multiprocessors are experimental and large multiprocessor Lisp applications are non-existent, soon both quality implementations and large programs will be available. Multiprocessors introduce constraints and challenges very different from those of uniprocessors. Section 8.3 reviews existing ideas about multiprocessor garbage collection and shows how a tool such as MARS can be helpful in better understanding the performance of such algorithms.

Many of the conclusions reached in this chapter are speculative. What processor speeds will be like in ten years, how large memories will grow, and how multiprocessors will be configured can only be projected based on existing technology. Nevertheless, intelligent guesses based on existing data can be valuable in helping designers plan for and build systems of the future. Of course, not all speculation proves immediately fruitful. In 1977, Peter Bishop wrote a PhD thesis exploring how Lisp running in a 48–64 bit address space should be configured [11]. In the 12 years since, address spaces have remained at 32 bits, but his thoughts and ideas may prove relevant in the near future.

In this chapter, I hope to answer some very basic questions about what algorithms will be appropriate to use and how well they will work. The need for incremental collection comes into question with faster processors. Generation collection, based on an assumption that most objects die young may break down if that assumption is invalidated. For multiprocessors, certain fundamental limitations prevent full utilization no matter what parallel algorithms are used.

8.1 Faster Processors

Making reasonable predictions about performance improvements in processors is difficult. Recent trends suggest that workstation CPU performance doubles every year or so, but sooner or later fundamental limitations will decrease the rate of further improvement. Furthermore, the designs that architects will use to achieve the higher performance (e.g., pipelined execution, a complex memory system, wide instruction words) are very difficult to predict. The CPU performance considered in this chapter is 100 times that of a Sun4/280, or approximately 1000 MIPS. Such systems are already being designed, albeit with very expensive technology. It is not unreasonable to believe that such systems will be in wide use by the end of the 1990's.

With such a processor (which I will refer to as the M1000), garbage collection performance considerations change dramatically. First, a faster processor implies faster allocation and more frequent, shorter collections for the same threshold sizes. Figure 8.1 shows the predicted pause lengths, frequencies and promotion rates for stop-and-copy garbage collection on the M1000 processor over a large range of threshold sizes. Pauses last only a handful of milliseconds, even for 16–32 megabyte thresholds. Collections are frequent, every 100–500 milliseconds. Promotion rates do not change from the M10 processor (our standard Sun4/280) since object lifespans (in terms of memory references) have not changed, and hence lifespans in absolute terms have become shorter.

The major constraint lifted by the M1000 processor is that collection pauses are no longer noticeable, even for large threshold sizes. If the physical memory is available, thresholds of 32–128 megabytes can be used without noticeable pauses. Alternately, even two megabyte thresholds can be used if the eight megabyte second generation also fits in the physical memory, since the second generation can also be collected without a noticeable pause. I conclude immediately that incremental collection is completely unnecessary with a fast enough processor. Incremental collection provides no advantage over stop-and-copy collection when processor speed is fast enough to eliminate noticeable pauses.

The only drawback of a faster processor running my test programs is that it requires more physical memory since either newspace must be enlarged to reduce promotion, or the second generation must fit completely in the memory. Both first and second generation

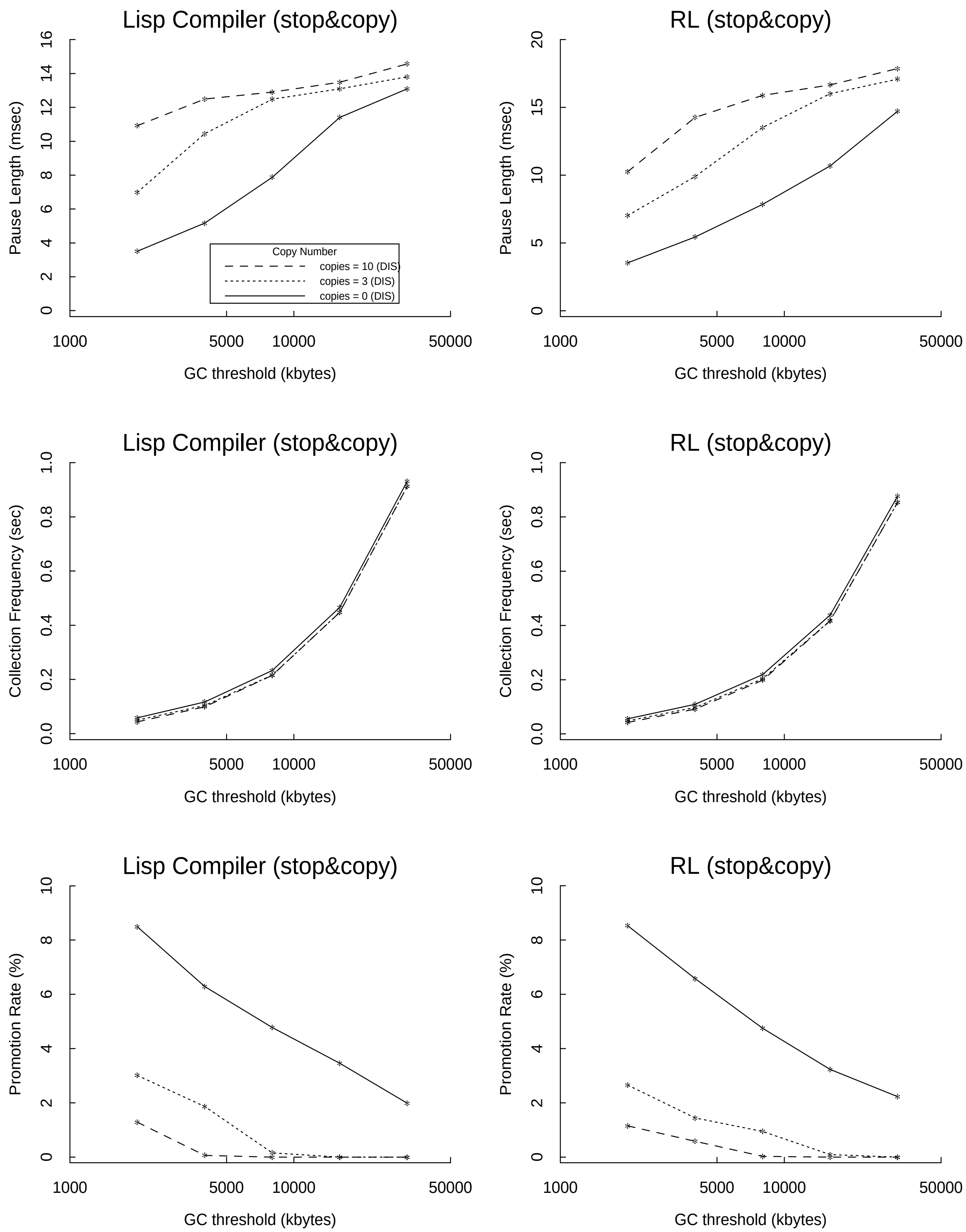

Figure 8.1: Predicted Performance for Two Applications. Predicted CPU speed is 100 times a Sun4/280.

collections are so frequent that any amount of paging during collection will greatly reduce performance. Faster processors appear to improve the effectiveness of garbage collection at the small cost of requiring additional memory. If the test programs used were the only programs being run on the M1000 processor, there would be little need for new or different collection algorithms.

8.2 Longer Lifespans

The four test programs measured take several hundred seconds to execute on the M10 processor and several seconds to run on an M1000 processor. Actual programs running on the M1000 processor are likely to take hundreds of seconds. If object lifespans increase as processor speed increases then the effectiveness of collection algorithms changes. If lifespans remain the same in terms of real time, then they are actually living for 100 times as many memory references on the M1000 processor. This prediction for lifespan behavior is not unreasonable. As I noted in Chapter 7, object lifespans may be tied to the length of a program subtask, which in turn may be tied to the process of interactive program development, where subtasks that take on the order of a second to complete are favored by developers. Some fraction of object lifespans may be relatively independent of processor speed. Validating such a prediction will require experience with faster processors.

If we assume that object lifespans increase with processor speed, the memory management of such programs becomes a critical problem. By using DIS and scaling the object lifespans by a factor of 100, I can predict the pause length, collection frequency, and promotion rates for the M1000 processor. Figure 8.2 shows the projected metrics for a wide range of threshold sizes up to 128 megabytes.

With an M1000 processor and increased object lifespans, the pause lengths associated with large thresholds are much longer than they were for the M1000 processor with normal lifespans. In this configuration, much of the allocated data is surviving even with very large thresholds. Even these longer pauses (in the tenths of seconds) are not noticeable until the newspace threshold is 100 megabytes or more. Newspace collections are also very frequent—more than once every four seconds even for the largest threshold sizes.

The newspace promotion rate associated with longer-lived objects is much higher, even for very large threshold sizes. Copying before promotion is effective for larger thresholds especially, but even with a copy number of 10 and a 32–megabyte threshold, 7% of the data allocated gets promoted to the second generation. For smaller thresholds and a lower copy number, promotion rates of 20–40% are not uncommon in the compiler. The high promotion rate causes the second generation to fill rapidly, even when the newspace threshold is large. Like the M1000 processor with normal lifespans, fast collections of the second generation are essential to performance.

Figure 8.3 shows the pause length, frequency, and promotion rate for collections of the second generation. Again, a wide range of threshold sizes is considered, up to one gigabyte. In this case, four newspace promotion rates are considered: 10, 20, 30, and 40%. The copy

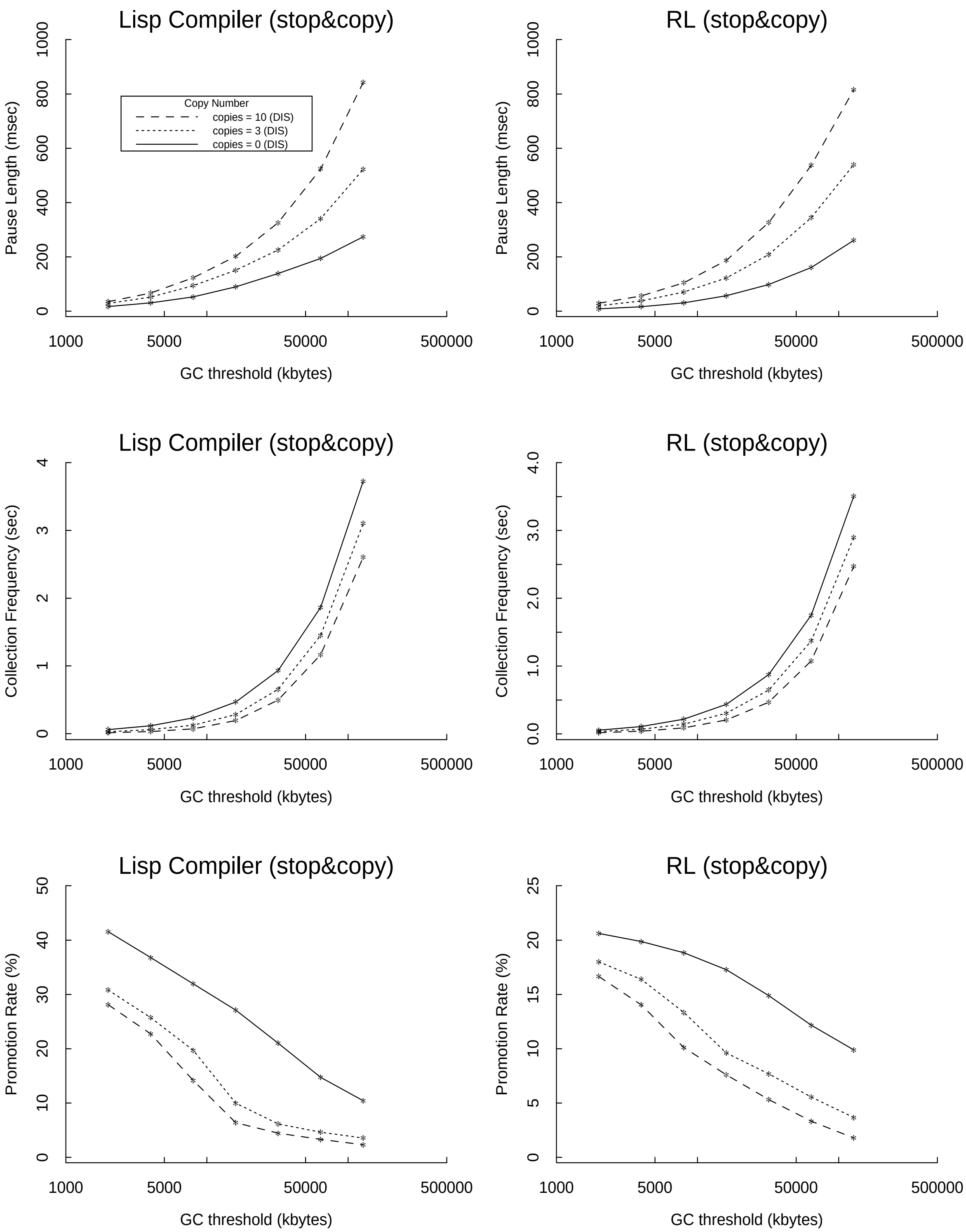

Figure 8.2: Predicted Performance for Two Applications. Predicted lifespans are 100 times those measured. Predicted CPU speed is 100 times a Sun4/280.

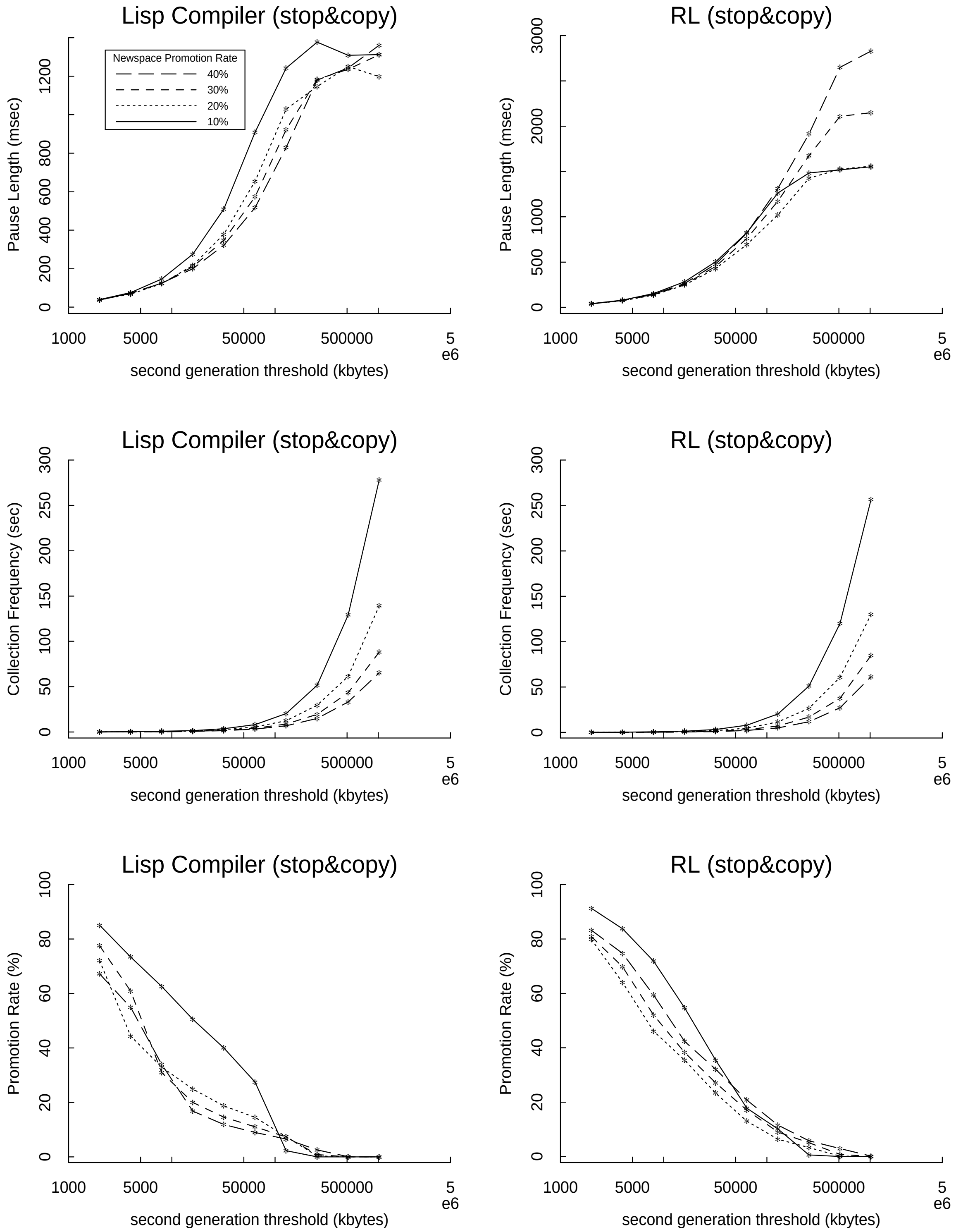

Figure 8.3: Second Generation Metrics for Four Applications Assuming Longer Running Programs and Faster CPU's. Object lifespan is assumed to be 100 times the lifespan actually observed.

numbers for the first and second generations are set at three.

Collections of the second generation typically take longer than collections of newspace, even for the same threshold size because second generation data tends to live longer than newly allocated data. Pauses lasting less than half a second require thresholds smaller than 64 megabytes. Second generation collections using thresholds of 64 megabytes and smaller occur every ten seconds or less, regardless of the newspace promotion rate. First generation promotion rate has relatively little effect on the second generation pause length and frequency metrics unless second generation thresholds are allowed to grow beyond 64 megabytes. For thresholds of 64 megabytes and smaller, the pauses are relatively undisruptive and frequent enough that the second generation must fit comfortably in the physical memory.

The promotion rates from the second generation to the third generation are much higher than from newspace to the second generation because the data in the second generation has already been alive for a while, and hence less likely to die before being promoted than newly allocated data. For threshold sizes of 64 megabytes, 15–30% of the second generation data is promoted to the third generation. As the first to second generation promotion rate increases, the fraction of data promoted to the third generation decreases. Data promoted with the 10% promotion rate has lived longer than data promoted with the 40% rate, and hence is more likely to continue to survive.

We also see that with a 128 megabyte threshold and a 10% newspace promotion rate, the second generation promotion rate drops dramatically. This drop occurs because the first two spaces at this point are sufficiently large to contain the program data for the entire lifespan of the program. The pause length with these parameters also flattens out.

From this analysis, it appears that even with a 10% newspace promotion rate, the second generation must be 32–64 megabytes to avoid extensive promotion to the third generation and to avoid noticeable pauses. Smaller thresholds will result in rapid promotion. Larger thresholds will be disruptive.

We can now look at the third generation, which, it appears, will be filling up fairly often. If we assume a 10% promotion rate from newspace to the second generation, we can consider second to third generation promotion rates of 20, 40, 60, and 80% (corresponding to second generation threshold sizes of approximately 100, 32, 7, and 3 megabytes, respectively, for the compiler).

Figure 8.4 shows the third generation pause lengths and frequencies in such a configuration. We again see that even for relatively large thresholds (i.e., 64 megabytes), collections are relatively frequent–more than once per minute. These pauses last up to a second or more, except in the cases where the threshold size is sufficient to contain the entire program in the third generation, in which case the pause length levels out.

The figure indicates that one strategy for the third generation is to make it sufficiently large that collections are infrequent. For example, with a second generation promotion rate of 20% and a threshold of one gigabyte, third generation collections occur every 15–25 minutes. A one second pause every 15–25 minutes is probably acceptable. The biggest

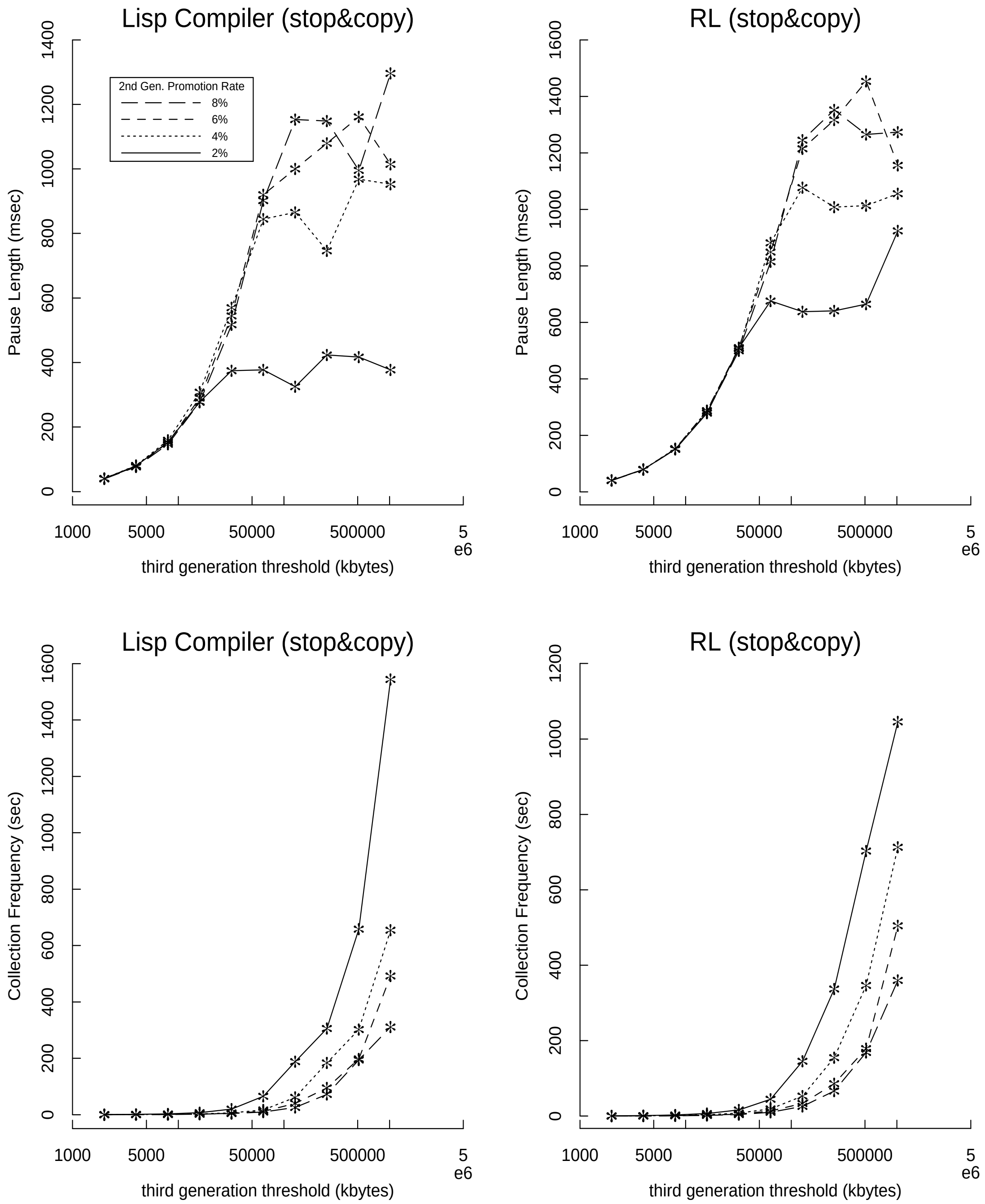

Figure 8.4: Third Generation Metrics for Four Applications Assuming Longer Running Programs and Faster CPU's. Object lifespan is assumed to be 100 times the lifespan actually observed.

problem with adopting a very large threshold size is the possible effect of paging during collection. The amount of paging caused by a collection of the third generation is related to the amount of live data being marked or copied. For large enough thresholds, increasing the threshold size may not significantly decrease the reference locality.

An alternative to using very large thresholds is to use a medium threshold size (e.g., 64 megabytes) and assume the third generation will also reside completely in the physical memory. This approach results in pauses of 0.4–0.9 seconds every minute or so. Tuning would be required to eliminate the longer (noticeable) pauses. Because the behavior in this case is so close to being unacceptable, there is some doubt that stop-and-copy collection would suffice at all.

In short, faster processors will require much more physical memory. In the programs considered, the first, second, and probably third generations have to have thresholds of approximately 64 megabytes each and fit entirely in the physical memory to prevent poor reference locality from limiting performance. Depending on the algorithm used, each generation will use 1.5–2.0 times the threshold size for its data. It would appear that the M1000 processor will need approximately half a gigabyte of physical memory if lifespans scale with processor speed.

With faster processors and longer lifespans, the problem of fitting the program into the physical memory becomes critical. A mark-and-sweep algorithm, with smaller working set and better reference locality, must be preferred in these circumstances. While memories are getting larger and cheaper per bit, the cost of memory is likely to always remain a large part of the overall cost of a computer system. Algorithms that require less memory will always be favored if their performance is roughly comparable to the best available performance.

Because we are looking at very large thresholds, the value of incremental collection again arises. We saw that pause lengths of one second and more occur with thresholds of greater than 100 megabytes. If physical memories large enough to support such large thresholds are available, then incremental collection becomes valuable by allowing very large threshold sizes without interactive disruption. From the simulations, however, there does not seem to be any clear evidence to indicate that two generations with 128 megabyte thresholds would be more efficient than four generations with 64 megabyte thresholds. Without a better understanding of the likely sizes of physical memories for the M1000 processor, the question of the value of incremental collection in such a system remains unresolved.

So far, we have considered one processor that is 100 times faster than current processors. The next section examines the performance of garbage collection algorithms on 100 processor multiprocessors.

8.3 Multiprocessors

The purpose of this entire chapter is to use MARS and DIS to explore the performance of garbage collection algorithms on machines and systems that do not yet exist. While commercial shared-memory multiprocessors do exist, production-quality multiprocessor Lisp systems are just now becoming available. Due to the lack of working multiprocessor systems and applications, much of what can be said about multiprocessor garbage collection is speculative.

There are many different performance aspects of multiprocessor garbage collection—enough to fill an entire thesis. The goal of this section is to outline areas of interest, noting when a tool such as MARS could be helpful is understanding the performance of multiprocessor garbage collection. After surveying current ideas in the are, this section focuses on the relationship between cache miss rate and bus utilization, and uses MARS to evaluate the potential speedup available using different collection algorithms. This section does not consider or propose new multiprocessor collection algorithms because without a test bed of real multiprocessor applications to use to evaluate the algorithms, the results obtained would be necessarily inconclusive. Throughout this discussion, only tightly-coupled shared-memory multiprocessors are considered.

8.3.1 The State of the Art

This section briefly reviews the current state of the art in multiprocessor Lisp systems, covering multiprocessor languages, collection algorithms, and potential performance bottlenecks. Several multiprocessing dialects of Lisp have gained prominence in the last five years. Multilisp [40], Qlisp [39], Mul-T [51], and SPUR Lisp [97] all extend Lisp or Scheme with features for multiprocessing. These languages all provide mechanisms to create independent threads of control that share a common address space. They differ mostly in the high-level features provided to make the programmer's job easier.

The ways in which control threads interact will profoundly affect the memory allocation and reference patterns of multiprocessor Lisp programs. Currently, there is no consensus on which programming paradigms are most effective for symbolic multiprocessing. This lack of consensus prevents accurate prediction of likely patterns of allocation and reference. Measurements of experimental systems, such as Multilisp [66], have been reported, but the experimental nature of the systems limits the usefulness of the results because large-scale commercial applications have never been ported and measured.

Recognizing how difficult it is to predict patterns of reference and allocation, the simplest assumption to make about control threads is that they act independently (i.e., never synchronize or share data). Independent control threads are the easiest to program because they require no inter-task communication and coordination. The "task-queue" paradigm of multiprocessing, where processors look for work from a central job queue, has processors acting in a relatively independent manner.

Like multiprocessor language extensions, multiprocessor collection algorithms have also been proposed. The algorithms fall roughly into two categories: "on-the-fly" algorithms, which allow collector processes and user processes (mutators) to execute concurrently, and "stop-and-cooperate" algorithms, which halt all processors during a collection and have them cooperate in reclaiming garbage. Dijkstra and others first proposed an "on-the-fly" algorithm as an interesting exercise in proving the correctness of parallel programs [28]. Lamport [53] and Kung and Song [52] provided modifications to the original algorithm to generalize it and make it more efficient. Newman, Woodward, and others modified and evaluated Lamport's algorithm using simulation [65, 64]. The Japanese Synapse project, which designed and built a multiprocessor Lisp machine, used a variation of Kung and Song's algorithm [58]. More recently, Japan's TOP-1 project [46] used a modified version of a real-time mark-and-sweep algorithm proposed by Yuasa [92]. Another approach to on-the-fly collection was proposed by Guy Steele [77]. Steele presents a practical design for multiprocessor mark-and-sweep collection that involves considerable distributed overhead.

On-the-fly collection has the advantage of providing real-time response because collection is constantly taking place. The disadvantage of the approach is that mutators and collectors must be careful synchronized to insure correctness. Dijkstra shows how difficult this synchronization is by admitting that he attempted to prove a flawed algorithm correct several times before finally discovering the correct algorithm. Collector/mutator synchronization may also require a significant overhead for operations that modify pointers. Unfortunately, the actual systems that have used on-the-fly algorithms have been experimental, and results from them are inconclusive.

Algorithms based on the "stop-and-cooperate" model are simpler to design and easier to implement. Butterfly Lisp used stop-and-copy collection where the root set was divided among the processors and each object was locked as it was being copied [1]. Multilisp also used a modified stop-and-copy algorithm [24]. Using incremental collection with multiple processors can provide on-the-fly-like response without the normal complex synchronization. In this configuration, processors stop and flip simultaneously, but then individual processors copy objects across the read barrier independently, as they reference the objects. By decoupling the copying from the flip, copying and allocation can go on simultaneously without complex synchronization. Because incremental collection introduces an overhead on every reference operation, the additional CPU overhead required to lock objects being copied is negligible, although extra space is required to store the locks.

This combination of incremental collection with multiprocessing was exploited by Appel, Ellis, and Li in an ML collector [2]. In their system, when mutators reference data in fromspace, they block and a collector process copies the data for them. Collector processes can also run concurrently with mutators, as long as they prevent mutators from accessing the pages they are scavenging.

None of the multiprocessor algorithms actually implemented uses generation collection techniques, although most could be extended to do so. As with on-the-fly algorithms, stop-and-cooperate performance results reported are for experimental systems, and their applicability to production-quality systems remains uncertain. Comparative results, where different multiprocessor collection algorithms on the same system are evaluated, are nonexistent. Hence, the current state of the art of multiprocessor garbage collection is quite primitive.

8.3.2 Multiprocessing and Bus Contention

We have already explored the data cache miss rates of different algorithms and programs using MARS. I have delayed until now an analysis of the *total* cache miss ratio (combining the miss rates of both the instruction and data caches). To compute the total miss ratio, the instruction fetch miss ratio is needed. Unfortunately, in its current configuration, MARS is unable to compute the instruction cache miss ratio for the test programs. Instead, I have used programs running on the BARB simulator to gather instruction cache miss rate results, shown in Figure 8.5.

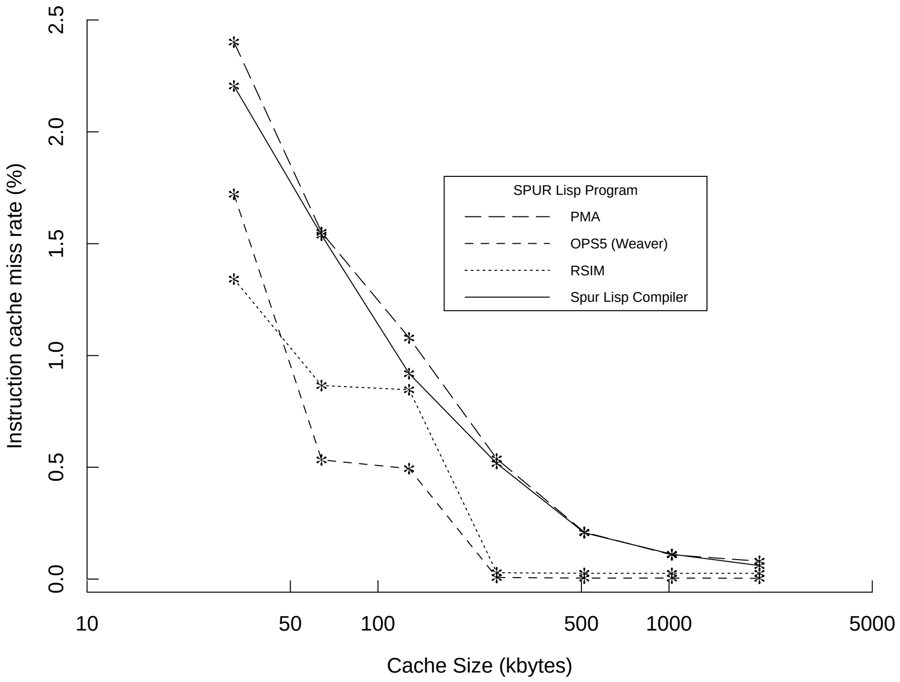

Figure 8.5: Instruction Cache Miss Rates for Four Applications.

The caches investigated were direct-mapped caches with 32-byte blocks and a write-back replacement policy. From the figure, we see that the instruction fetch miss rates are much lower than the data fetch miss rates, especially for large caches. The figure shows that the miss rate curves are quite similar despite large differences in the test programs. Averaging the miss rates for each cache size, I can obtain an approximate instruction cache miss ratio to combine with the data cache miss ratios collected using MARS to obtain an approximate total miss ratio for each MARS test program and collection algorithm. Another approximation made in combining the values is the assumption of a fixed ratio of data references to instruction fetches. From Table 3.1 in Chapter 3, we see that the ratio of heap to instruction references is approximately 7:1 and relatively constant for each test program (ranging from 6:1 to 10:1).

Because instruction references are so frequent and hit so often, the total miss rate is much smaller than the data cache miss rate. Figure 8.6 shows the total cache miss rate for mark-and-sweep and stop-and-copy collection with three large cache sizes. In most cases, mark-and-sweep shows a better total miss rate than stop-and-copy collection. In all cases, the total miss ratio is less than 0.6%. The effect of total miss rate on execution time will be negligible if the programs are run on a uniprocessor since a 1% miss rate corresponds roughly to a 10% increase in execution time.

On a multiprocessor with a memory bus shared by the processors, the bus is a critical resource divided between all processors and a potential bottleneck to performance. In such systems, the bus is often one of the resources that limits the possible degree of multiprocessing available. To see how the total miss rate affects potential speedup, a simple MM1 queueing model can be used. An MM1 queueing model is one that assumes exponentially distributed arrival and service times. If the tasks executing on each processor act independently, the bus request workload is distributed exponentially. The model used assumes a 32-bit wide bus, that the bus and processor cycle at the same rate, and that the service time for a cache miss is eleven bus cycles: eight cycles to deliver a 32-byte cache block and three to setup the request (the latency before data is transferred). These assumptions correspond to the design of the SPUR bus architecture. The total miss rate corresponds to the fraction of requests that require a bus transaction.

Figure 8.7 shows the possible parallelism available for multiprocessors with different numbers of processors given that each processor has a particular total miss rate. With a miss rate as low as 1.5%, the possible parallelism is limited to approximately seven processors due to bus saturation. To achieve near linear speedup beyond 30 processors, the total miss rate must be kept below 0.3%. We have seen that for large caches, the instruction cache miss rate can be lowered to well below 0.3%. The data cache miss rate is the metric that limits our potential speedup. To fully utilize multiprocessors with 30 processors or more, programs must show extremely good locality of reference in the data cache. In Chapter 6, we saw that proper choices of algorithm, threshold size, and data cache size can provide low data cache miss rates.

We have already seen the total miss rates of mark-and-sweep and stop-and-copy collection for several large test programs. With our queueing model, we can now see how the total miss rates translate into potential speedups. This analysis is only accurate if we assume that the different processors on a multiprocessor are executing independently. While such an assumption is simplistic, the lack of more specific data makes it necessary.

Figure 8.8 shows the potential speedup curves given several threshold sizes for mark-and-sweep and stop-and-copy collection, assuming that the instruction and data cache sizes are fixed at one megabyte. The figure shows the potential user program speedup by reducing the curves by the garbage collection overheads measured with MARS. Because mark-and-sweep collection generally provides lower miss rates, the potential speedup is higher than for stop-and-copy collection. Furthermore, smaller threshold sizes fit better in a one megabyte cache,

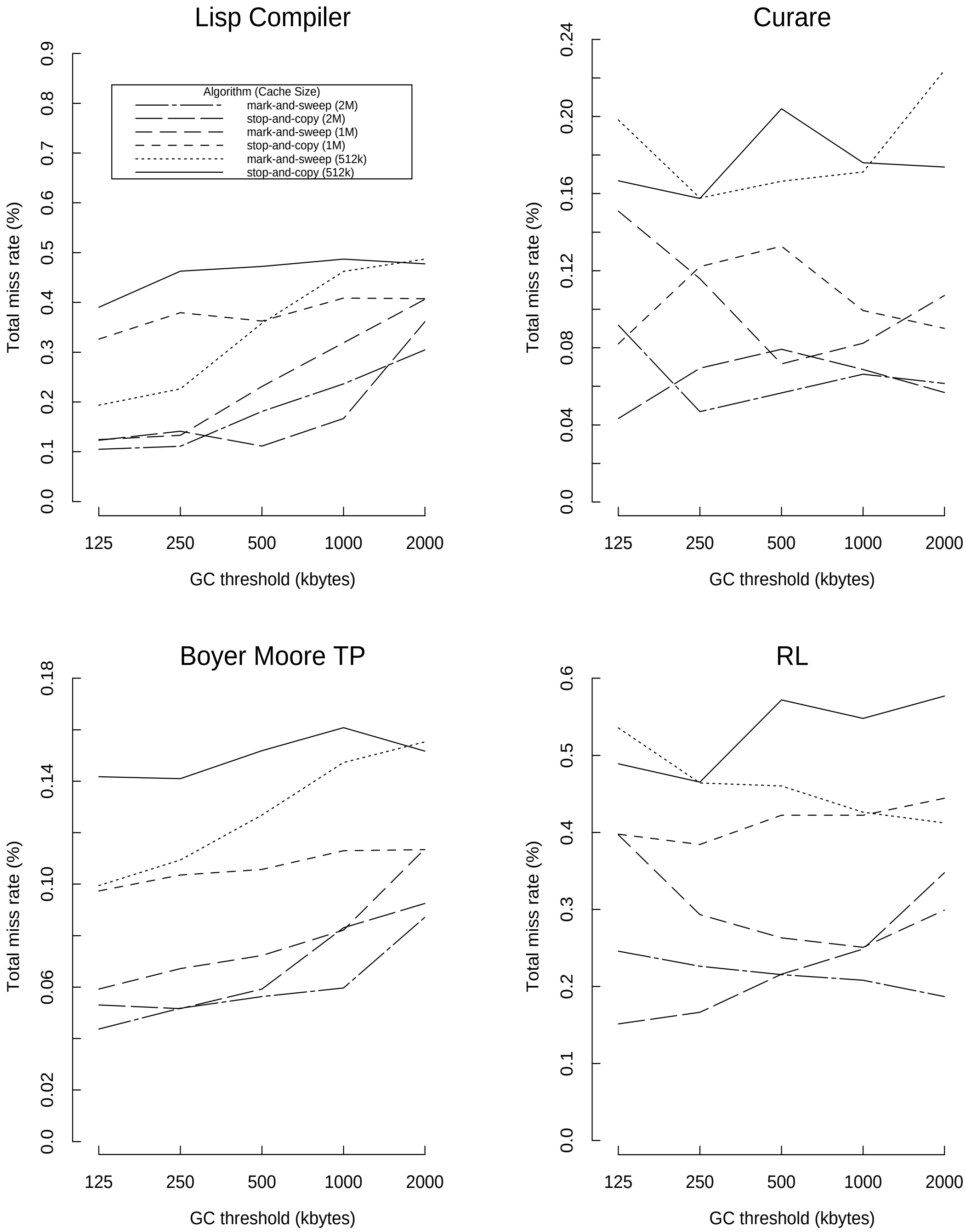

Figure 8.6: Total Cache Miss Rates for Three Collection Algorithms.

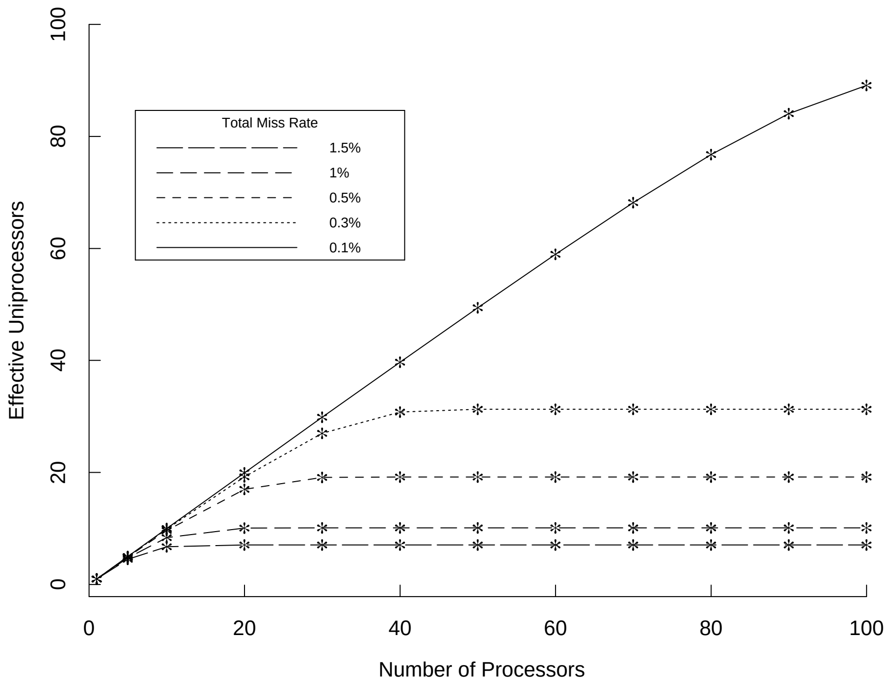

Figure 8.7: Maximum Effective Uniprocessors for Different Miss Ratios.

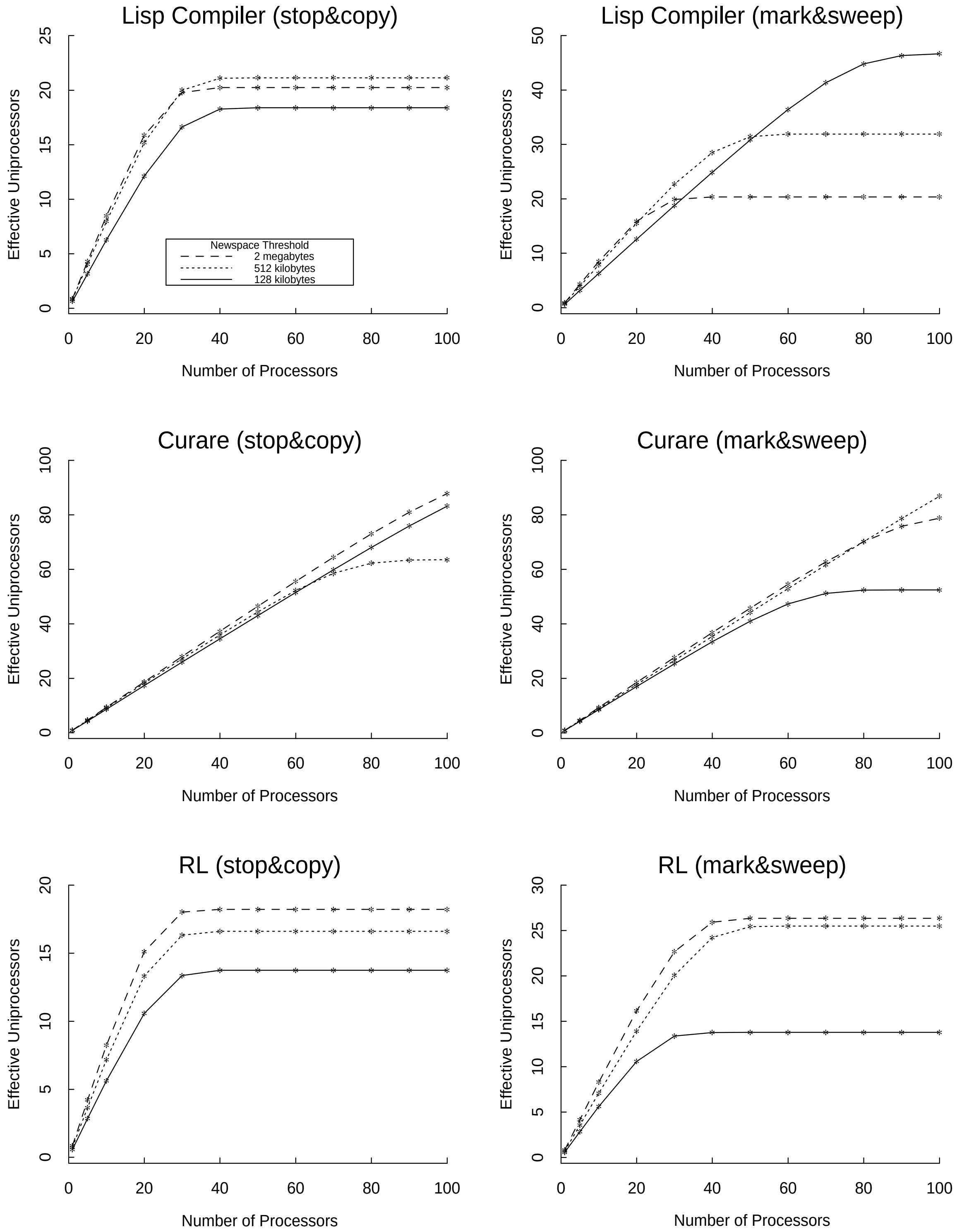

Figure 8.8: Maximum Effective Uniprocessors for Different Algorithms and Threshold Sizes. Garbage collection induced overhead is taken into account.

and so smaller thresholds tend to show better speedups. Predictably, the increased overhead caused by the smaller thresholds sizes lowers the potential benefit of these configurations. Nevertheless, for programs such as the compiler, the lower miss rate provided by the smaller threshold more than makes up for the additional overhead.

Smaller thresholds do not always produce larger speedups, however. Because the 125 kilobyte threshold causes the mark-and-sweep algorithm to promote more data, the 125 kilobyte threshold shows lower maximum speedups in Curare and RL. Tuning, by way of increasing the copy number to reduce promotion, would probably result in higher performance for the small thresholds. Figure 8.8 shows the magnitude of the potential benefit of lowering the data cache miss rate on a multiprocessor. In the compiler, using mark-and-sweep collection can increase the potential speedup by more than two times. With tuning, garbage collection can be used to greatly enhance potential speedups in a shared-memory multiprocessor.

Chapter 9

Conclusion

This thesis investigates two distinct fields of computer science: simulation and garbage collection. The comparative evaluation of garbage collection has been used to demonstrate the effectiveness of a general simulation technique, object-level tracing, which can be used to explore a wide range of algorithms and implementations. Just as address-level, trace-driven simulation is an effective tool for predicting the performance of cache and virtual memory systems, object-level tracing may eventually be commonly used to evaluate programming language implementations. In particular, the development of sophisticated runtime systems will be greatly facilitated using tools like MARS.

Throughout this thesis, simulation techniques and collection algorithm performance measurements have been intermingled. In this chapter, I separate them.

9.1 Simulation

Simulation as an evaluation tool has had mixed success through the history of computer science. Useful simulations require precise duplication of the system being simulated, which in turn often requires extraordinary computer resources. Simulation of computer systems is especially resource-intensive, since the simulation often runs hundreds of times slower than the system being simulated, and performance evaluation can require simulating hours or days of execution. Being as resource intensive as it is, simulation must provide accurate results to be of value. Programs that represent the actual workload that will be placed on a system must be simulated.

MARS was designed to provide accurate results and at the same time allow a rapid exploration of the space of design parameters. MARS has obvious strengths. Because MARS is attached to a commercial Common Lisp system, simulations are driven by actual programs and new workloads are trivial to evaluate. New collection algorithms can be designed and incorporated into MARS in less than one week. Many algorithm parameters can be changed just by modifying one line in an input file. MARS also provides a large amount of performance information about the algorithms it is simulating, including CPU overhead and reference locality measurements.

Overall, I believe that MARS is an important tool that has demonstrated the feasibility of using simulation to evaluate runtime systems. But along with its strengths, MARS has weaknesses. It is very resource intensive—simulation slows execution by 100 times and doubles the physical and virtual memory requirements of the simulated program. In particular, measuring stack locality through stack simulation incurs a hefty overhead on every memory reference. Because the locality information provided by MARS is important, turning off stack simulation is not an acceptable option. Partial stack simulation can be used to cut the cost of stack simulation in half, but a large overhead remains.

DIS was created solely because MARS slowed programs too much to measure the long-term behavior of collection algorithms. Because long-term behavior is of critical interest, MARS must either be abandoned for this task, or accelerated significantly. While the initial implementation of MARS is not highly optimized, its performance cannot be improved by a factor of ten without using new hardware, such as a multiprocessor or perhaps a hardware implementation of stack simulation. However the performance is improved, MARS would benefit greatly if it only slowed test programs by a small factor (e.g., two to five times).

Another drawback of MARS is its ability to measure almost anything. While this would appear a benefit, in fact I sometimes found myself lost in a sea of data collected while using it. Being able to measure all aspects of performance has the drawback that the relative importance of different aspects becomes less clear. This thesis has attempted to narrow the possible metrics to reach interesting conclusions, but there are many other relationships (e.g., the relationship between object size and lifespan), not explored in this thesis, that might also have been interesting. The ability to collect data does not guarantee the ability to reason about what has been collected.

Nevertheless, some of the results generated using MARS were surprising and informative. For example, the overall effectiveness of generation mark-and-sweep collection was unexpected. The locality behavior of incremental collection, not largely different from stop-and-copy collection, was not anticipated. The modified eq method of implementing incremental collection also appears promising. By providing non-intuitive insights, MARS has proven itself to be a valuable tool.

MARS also provided information about complex algorithms that are otherwise difficult to implement and measure. Research in garbage collection has been going on for nearly thirty years but little comparative evaluation has been attempted. Exploratory measurement using a tool like MARS appears to be a convenient and cost-effective way to investigate new algorithms. MARS will also be useful in exploring runtime system features other than garbage collection. Simulators like MARS are an enabling technology that opens the doors for the design and implementation of more sophisticated and effective runtime system technologies.

9.2 Garbage Collection

Much of this thesis has focussed on the performance of garbage collection implementations. With MARS, I have examined the effectiveness of known algorithms to provide more insight into their performance. In the preceding chapters, I reached definite conclusions about performance based on simulation. I hope that the conclusions presented here will encourage implementors to incorporate the ideas into actual systems and prove or disprove the validity of my results.

The most important result, echoed in many of the chapters, is that generation mark-and-sweep collection, as described in this thesis, is significantly more effective than copying collection. For an allocation threshold of 500 kilobytes, which results in non-disruptive pauses, the algorithms compare in the following way. The total CPU overhead of mark-and-sweep collection is 1.5–4.7% greater than the overhead of copying collection, mostly because of the threshold-independent cost of sweeping and the additional cost of indirect vector references. At the same time, the physical memory needed by copying collection is approximately 30–40% larger than the memory needed by mark-and-sweep collection to achieve the same page fault rate (ten faults per second). Furthermore, the mark-and-sweep algorithm often shows a lower cache miss rate, although the relative performance varies widely with threshold size and cache size. The compacting effect of a copying collector, important before generation methods reduced the memory demands of garbage collection, offers no advantage if the entire newspace resides in memory. The greatest disadvantage of the mark-and-sweep algorithm is its use of en masse promotion, which for a 500-kilobyte threshold and a copy number of three results in 25–75% more promotion than copy count promotion (promoting 9–12% of all objects allocated). Chapter 7 showed the negative effects of a high promotion rate, both from decreased reference locality and from additional second generation collections. En masse promotion is most natural for mark-and-sweep collection, but copy count promotion is also possible if several bits per object (much like the mark bitmap) are reserved to record a per-object copy count. The performance of this solution needs to be explored further.

One reason for the continued use of copying collection is that the Baker incremental collection algorithm requires copying between semispaces. This thesis attempted to evaluate the performance of incremental collection on stock hardware. At the outset of this investigation, I was expecting to find that Baker-style incremental collection was not cost-effective on stock hardware. But in this thesis, I have proposed and investigated two implementation methods of this algorithm that show low overheads (from 11–27% with reasonable operating system support) when threshold sizes are large enough. In the future, I hope to implement these methods to determine their actual performance. One aspect of incremental collection that I once considered a potential problem—its reference locality–was by my measurements not a problem at all.

Because incremental collection allows arbitrary growth of threshold size, these algorithms are more important as physical memories grow. Mark-and-sweep algorithms, on the other hand, become more valuable as applications place a greater demand on the available memory. In the future, both memories and applications may become so large that a hybrid incremental mark-and-sweep algorithm will be required. Developing such an algorithm is a challenge for the future.

9.3 Future Work

One logical direction for future work is to evaluate new algorithms using MARS. To make the workload for this thesis manageable, I reduced the algorithms considered to three basic ones. Many variations of these algorithms exist, as do many different algorithms. Of particular interest are variations of mark-and-sweep collection, which showed promise in this research. Yuasa proposed a realtime mark-and-sweep algorithm [92]. Extending the algorithm with generations and examining its performance would be valuable. Several proposals for non-copying generation mark-and-sweep algorithms exist (although nothing has yet been published about them). By entirely avoiding copying, the possibility of fragmentation arises. MARS could be used to determine the overall reference locality, and the space losses due to fragmentation in these algorithms.

Because premature promotion is a potential problem, several solutions to the problem should be explored using MARS. First, Ungar's demographic feedback promotion policy should be investigated to determine its effectiveness for Lisp programs. Second, algorithms that demote data (i.e., move it into a younger generation) might be effective in increasing reference locality and reducing the number of write barrier traps. Also, techniques for maintaining copy counts, such as Shaw's bucket brigades should be more carefully evaluated.

Along with investigating new algorithms, MARS itself should be extended and optimized. The locality of program and collector references should be separated to allow a more complete understanding of the locality effects of algorithms (i.e., some may show better collector locality, while others better program locality). If the performance of MARS is substantially improved, longer runs would provide more information about longer-lived data. The value of MARS as a tool is also related to the number of potential users who can benefit from its information. If the performance of MARS is improved, smaller machines with smaller memories would be able to run MARS, thus enabling more users to use it.

An alternative to substantially changing MARS is to upgrade DIS to use more information about program execution, thus making its analysis more accurate. By collecting a series of lifespan distributions throughout the execution of a program and feeding all of them into DIS, time-dependent behavior could be more accurately reproduced.

In the future, MARS could be improved by redesigning the way that collection algorithms are specified. Currently, the algorithms are just written in C, compiled, and linked with the MARS object code. A better approach would be to design a simple, stylized language for describing collection algorithms and build a translator that translates this language into efficient C code. This language would further simplify and facilitate the rapid development and evaluation of new collection algorithms.

Other avenues of research are inviting. If a production-quality multiprocessor Lisp implementation was available, MARS could be used to evaluate the effectiveness of multiprocessor garbage collection algorithms. In addition to the data that MARS already collects, including CPU overhead, pause length, and reference locality, a version of MARS simulating multiprocessor garbage collection could be used to evaluate additional performance tradeoffs.

In particular, different allocation strategies could be compared. Allocation can be a source of contention if memory is allocated from a central pool. A better strategy allocates medium-sized chunks of memory from a global pool to individual processors and gives the processor exclusive access to the chunk once it is allocated. The effect of this strategy on locality of reference, as well as the need for such a strategy could easily be determined.

Another important policy that needs investigation is the locking strategy employed by processors that are cooperatively collecting. Different granularities of locking should be examined and the contention for the locks should be measured. Fine-grained locks, as used in Multilisp and Butterfly Lisp, require more space but reduce contention. The necessity of per-object locks should be determined. Traversal strategies that minimize contention should also be designed and evaluated. The great advantage of using a simulator to evaluate these policies is that a simulator eliminates at least some of the complexity associated with a multiprocessor garbage collection implementation. Practical experience has already shown that efficient and correct implementations of policies on real multiprocessors is challenging and time-consuming.

MARS could also be used to look at the tradeoffs between on-the-fly, stop-and-cooperate, and incremental multiprocessing collection. On-the-fly collection is complex, but offers potential real-time response. MARS could used to determine the exact overhead of this approach and help us decide if the improved response balances the additional overhead. Incremental multiprocessor collection intermixes collector references and program references. On a uniprocessor, we have seen that little dilution of reference locality occurs with incremental collection. MARS could be used to determine if this intermixing actually improves collection speedup by eliminating a potential bottleneck caused by poor locality during a cooperative garbage collection.

Throughout this thesis, I have suggested a that tool like MARS could be used to explore runtime system design alternatives. I believe that as CPU's become faster, runtime systems will become more sophisticated for two reasons: first, because faster CPU's will impose harsh constraints on an executing program (e.g., demand a fast memory system or require the execution of multiple instructions per cycle). An adaptive, dynamic runtime system may be necessary to meet these constraints and isolate the user from them. The details of meeting the constraints can be hidden from the programmer by the runtime system, just as the details of machine instruction sets are hidden from programmers by high-level languages. Second, software development has become a critical bottleneck in the evolution of computer systems. Very-high-level languages, which hide many of the details of their implementation, are required to solve this problem. Adaptive runtime technologies (ART), which can allow high-level user abstractions and efficient implementation of those abstractions will be necessary. Garbage collection is a perfect example of a work of ART. Garbage collection gives the user an illusion of an infinite memory from which he can freely allocate, while at the same time reorganizing the program objects so that they fit well in the available memory. The cost of garbage collection is 5–20% of the CPU. With increasingly fast processors, many users will gladly trade some performance for a simplified programming model.

Possible components of adaptive runtime systems include dynamically compiled, linked, and relocated libraries, automatic procedure integration and optimization, automatic execution profile collection and system decision making based on the collected data. MARS will be valuable in the future because it will provide a means of rapid evaluation of these sophisticated systems. Currently, garbage collection algorithms are generally considered esoteric and obscure, if also valuable. But only through such esoteric and sophisticated methods can the full power of complex future computer systems be made available to large numbers of unsophisticated computer users. MARS is one enabling tool that can allow this breakthrough to occur.

Appendix A

Instruction Sequences

This appendix contains the instruction sequences mentioned in Chapter 5. Various SPARC implementations of the write barrier and read barrier are presented. The instruction sequences in this appendix contain SPARC assembly code in the recommended style, also using the pseudo-instructions defined in Appendix E of the SPARC Architecture Manual [79].

RISC instruction sequences can be confusing, especially so because of delayed transfers of control. The SPARC architecture has two kinds of delayed branches. The normal kind executes the instruction in the delay slot whether or not the branch is taken. The second kind, called an annulled branch, only executes the instruction in the delay slot when the branch is taken. If the branch is not taken, the instruction is not executed, and a nop occurs for that cycle. Annulled branches are indicated with "branch,a" syntax in SPARC assembler.

A.1 The Hardware Write Barrier Trap Handler

First, the code sequence for trap handling suggested by Johnson is presented [47]. Without a few hardware extensions, Johnson estimates that a user trap interface in SPARC which preserves state, recovers operands, invokes the user trap, and returns, requires 106 instructions. John suggests extensions that include five special registers to record the opcode, operands and the destination register of a trapping instruction. Opcodes must also be added to read and write these new registers (the `rd` and `wr` instructions). With these new instructions, the write barrier trap handler interface simply recovers the operands and invokes the handler. Because the trap re-executes the trapping instruction and does not need to update the destination register, the write barrier trap return is simpler than Johnson's more general handler. With his recommended hardware extensions, the write barrier trap interface sequence is reduced to the sequence in Figure A.1. This sequence requires six instructions to execute, plus an additional delay instruction when the trap is taken, resulting in a total of seven instructions per trap (i.e., $C_{trap} = 7$). The same sequence can be used to implement a hardware read barrier trap handler.

```
# read the object register
rd OP1, %object

# read the container register
rd OP2, %container

# read the program status word and enable user traps
rd PSR, %psr
or %psr, UTRAP_ENABLE, %tmp

# begin write barrier trap (i.e., word marking) here
.
.
.

# return to the program, restoring the program status register

jmpl %o7,%0
wr %psr, PSR
```

Figure A.1: Extended SPARC Instruction Sequence for a Hardware Write Barrier Trap Handler

A.2 Handling Write Barrier Traps

This section presents a SPARC instruction sequence for word marking as proposed by Sobalvarro [75], who originally presented it in MC68020 code. The word marking algorithm uses a two-level table structure to mark words of memory that contain intergenerational pointers. The memory is divided into 64-kilobyte segments. The first-level table uses a byte to indicate if a segment contains any intergenerational pointers. This table, called the segment modification table (SMT), contains 16 kilobytes of entries to map a complete 32-bit address space. Each segment has a corresponding bitmap that indicates which locations in the segment contain intergenerational pointers. Each bitmap is called the modification-bit table (MBT) for the segment, and the table mapping segments to MBTs is called the MBT map. Each bit in a segment's MBT maps to a 4-byte word in the segment. A bit of the MBT is set if the address it maps contains an intergenerational pointer.

When an intergenerational pointer is created, the corresponding byte of the SMT and bit of the segment's MBT must be set. Figure A.2 shows the SPARC instruction sequence for this action. The following assumptions are made:

1. The Lisp system constants vector, which contains a pointer to the SMT and a pointer to the MBT map, is stored in the global register `%g_sysconsts`.

2. The object containing the intergenerational pointer is in register `%container` and the offset of the pointer in the container is in `%offset`.

This code sequence is very similar to the sequence provided by Sobalvarro, except that more instructions are needed because the SPARC lacks indirect addressing and bit operations. The cost of marking a word using this sequence is 16 instructions ($C_{wbhandler} = 16$). While a variety of temporary register names are used in the code, only four temporary registers are needed at any one time.

A.3 Implementing the Write Barrier with Software Tests

Implementing the write barrier with software tests requires an inline test on each pointer store and a function that is called for further tests. The following assumptions are made:

1. The boundary between newspace and oldspace is stored in the global `%g_g01boundary`.

2. The Lisp system constants vector, which contains the boundaries between the other generations, is pointed to by the global `%g_sysconsts`.

3. The pointer being stored is located in register `%object`.

4. The object being stored into is located in register `%container`.

```
markWord:
  # first, get the segment number of the pointer's address
  mov %container + %offset, %segnum
  sll %segnum, 16, %segnum

  # get a pointer to the segment modification table (SMT)
  ld %g_sysconsts + SMToffset, %smt
  # set the byte in the SMT
  stb -1, %smt + %segnum

  # get a pointer to the MBT map, then get the MBT
  ld %g_sysconsts + MBTmap_offset, %mbtmap
  srl %segnum, 2, %segindex
  ld %mbtmap + %segindex, %mbt

  # calculate the offset in the mbt in bits
  mov %container + %offset, %segoffset
  and %segoffset, 0xffff, %segoffset

  # convert byte address to bitmap offset (1 word / bit)
  srl %segoffset, 2, %bitoffset
  # calculate the byte offset in the MBT
  srl %bitoffset, 3, %byteoffset

  # get the byte from the MBT
  ldb %mbt + %byteoffset, %bytecontents

  # calculate the bit offset in the byte
  and %bitoffset, 3, %bitinbyte
  # create a mask to ``or'' with the contents
  sll 1, %bitinbyte, %bitmask
  or %bytecontents, %bitmask, %bytecontents
  stb %bytecontents, %mbt + %byteoffset
```

Figure A.2: Instruction Sequence for Word Marking

```
  # pointer store without write barrier check:
  st %object, %container + offset
```

```
  # pointer store with write barrier check:
  # first, check for a container in newspace.
  cmp %container, %g_g01boundary

  # test, performing the store in the delay slot (with annulling)
  bgu,a done
  st %object, %container + offset

  # here, the container isn't in newspace, so setup a
  #      call to a function that checks other possibilities
  # move one of the arguments in the delay slot of the call
  mov %object, %o0
  mov %container, %o1
  call writeBarrierTest
  mov offset, %o2

  # after the call, we are ready to perform the store
  st %object, %container + offset

done:
```

Figure A.3: Software Test Write Barrier Instruction Sequence

The test sequence is shown in Figure A.3. This instruction sequence results in three additional instructions being executed when the container is in newspace ($C_{newstorep} = 3$). If the container is in oldspace, the cost of the sequence is eight additional cycles plus the cost of the function `writeBarrierTest`. That function is shown in Figure A.4. Because we know that intergenerational stores are very infrequent, most of the time that `writeBarrierTest` is called, it simply returns. This will happen if the object is an immediate or if the generation of the object is greater than or equal to the generation of the container. Most often, the generation of the object is 3, indicating it is a system object. A large majority of the time, the function will exit at either return A (costing five cycles) or return B (costing nine cycles). I estimate the cost of this function, assuming `markWord` is not invoked, as the average of cost A and cost B, or seven cycles. The total cost of a non-marking oldspace store is the sum of the function call setup and the function call cost, or $8 + 7$ cycles (i.e., $C_{oldstorep} = 15$).

If `markWord` is invoked, the cost is the cost of `markWord`, estimated above, plus the additional overhead of calling `writeBarrierTest` and determining if the word needed marking. If we assume the most common cause of a write barrier trap is storing a pointer to a newspace object into an oldspace container, then the test at branch C is the most common path through the function. The cost of this path is the cost of calling `writeBarrierTest` (eight cycles) plus the cost of getting to branch C (11 cycles), plus the cost of `markWord` (16 cycles). The total cost of a software write barrier trap is approximately 35 cycles ($C_{wbtrap} = 35$).

A.4 Implementing the Read Barrier with Software Tests

The read barrier can also be implemented with software tests around every pointer load. The code is divided into a test that is placed inline with each load and a function that handles object transportation when it is required. The following assumptions are made:

1. The boundaries of fromspace are stored in globals `%g_fromspaceBase` and `%g_fromspaceTop`.

2. Transported objects are copied to the location specified by the global register `%g_copy`.

3. A pointer is being loaded into register `%dest`

The inline test is shown in Figure A.5. If the loaded pointer does not point into fromspace, the one instruction sequence is replaced with either a four instruction or a seven instruction sequence. Assuming they are both equally likely, the average added cost of $(3 + 6)/2$ cycles is assumed ($C_{loadp} = 4.5$). There is also a small chance that the loaded pointer will be a fixnum whose value is equal to an address in fromspace. Although we must test for this, it is so unlikely that it does not affect the estimated cost. When a pointer does point into fromspace, the function `readBarrierMove` is called at a total cost of nine cycles, including the tests. In addition, time is spent in the function.

The function `readBarrierMove` is shown in Figure A.6. The added cost of this function,

```
  # assume %object == %o0, %container == %o1, offset = %o2
  # the notation g(%object) means ``generation of'' %object
  # we don't ``save'' because we don't need new registers
writeBarrierTest:

  # first, check if %object is a fixnum (immediate)
  # assume the tag is in the lower 2 bits, and that a tag
  #       of #b00 means a fixnum, which doesn't require marking
  and %object, 3, %o_tmp
  bnz next1
  nop
  retl                         # return A: type(%object) == fixnum
  nop

next1:
  # if (g(%object) == 3) return
  ld %g_sysconsts + offset_g23, %o_g23boundary
  cmp %object, %o_g23boundary
  bgu next2
  nop
  retl                         # return B: gen(%object) == 3
  nop

next2:
  # if (g(%object) == 0) mark word
  cmp %object, %g_g01boundary
  bgu markWord                 # branch C: gen(%object) == 0
  nop

next3:
  # if (g(%container) == 3) mark word
  cmp %container, %o_g23boundary
  blu markWord
  nop

next4:
  # g(%object) == {1,2}, g(%container) == {1,2}
  # if (g(%object) == 1) then
  ld %g_sysconsts + offset_g12, %o_g12boundary
  cmp %object, %o_g12boundary
  blu gobjecteq2
  nop

  #   if (g(%container) == 2) mark word
  cmp %container, %o_g12boundary
  blu markWord
  nop
  #   else return
  retl
  nop

gobjecteq2:
  # since g(%container) <= 2, return
  retl
  nop

markWord:
  <see previous figure>
  retl
  nop
```

Figure A.4: Complete Write Barrier Test Function

```
  # pointer load without read barrier check
  ld %object + offset, %dest
```

```
  # pointer load with read barrier check
  ld %object + offset, %dest

  # next, test against the bottom of fromspace
  cmp %dest, %g_fromspaceBase
  blu done
  nop

  # test against the top of fromspace
  cmp %dest, %g_fromspaceTop
  bgu done
  nop

  # here, the new pointer points into fromspace
  # call a function to relocate the object and
  #      update the pointer (the new pointer is in %o1)
  call readBarrierMove
  mov %dest, %o0

  mov %o1, %dest
done:
```

Figure A.5: Software Test Read Barrier Instruction Sequence

```
  # assume %object == %o0, the pointer into fromspace
  # we want to relocate it and return the new pointer in %o1

readBarrierMove:

  # first, check if the new pointer is really an immediate
  and %dest, 3, %o_tmp
  bnz more
  nop
  retl
  mov %o0, %o1

more:
  # next, check if type(%object) is a cons (most common)
  # assume the tag is in the lower 2 bits, and that a tag
  #     of 3 means a cons
  and %object, 3, %tmp
  cmp %tmp, 3
  bne nonCons
  nop

moveCons:
  ld %object-3, %tmp
  st %tmp, %g_copy
  ld %object + 1, %tmp
  st %tmp, %g_copy + 4
  mov %copy + 3, %o1

  retl
  add %copy, 8, %copy

nonCons:
  et cetera
```

Figure A.6: Read Barrier Relocation Function on top of the cost of transporting the object, which must be done anyway, is five cycles, assuming that the pointer is almost never an immediate. This brings the total cost of a read barrier trap not counting the object transport to nine cycles to setup the call to **readBarrierMove** and five cycles in the function. That is, the cost of a software read barrier trap is 14 cycles ($C_{rbtrap} = 14$).

A.5 Allowing Fromspace Aliases and Modifying Eq

If the read barrier is implemented by allowing fromspace aliases and modifying the eq operation, every eq test will require the instruction sequence in Figure A.7. The cost of this sequence depends on whether either of the pointers points into fromspace. Assuming they are not eq and neither points into fromspace, which is the most common case, then they are both likely to point either above or below fromspace (into tospace). The relative positions of fromspace and tospace change frequently, so which test fails is likely to average. The estimated cost of this sequence is 12 instructions ($C_{eqfalse2unk} = 12$).

In addition, since fromspace aliases may not be written to memory, in read barrier implementations that allow fromspace aliases in registers, all pointer stores must be modified to the instruction sequence in Figure A.8. The average cost of this sequence is three instructions ($C_{allstorep} = 3$).

A.6 Estimating Collection Overhead

To estimate the overhead of different garbage collection algorithms, we must look at the inner loops of the collection algorithms and count the instructions required to perform each basic operation. Event counts provided by MARS can then be combined with the instruction counts to arrive at the overall cost in cycles. Because different algorithms require different actions during allocation, I include the allocation cost in the overall cost of each algorithm. Because the incremental and stop-and-copy algorithms are so similar, only the stop-and-copy and mark-and-sweep algorithms are considered in this appendix. While these algorithms are described in pseudocode in another appendix, the pseudocode and instruction sequences in this appendix differ because the instruction sequences are tuned to avoid any unnecessary overhead.

A.6.1 Copying Collection

Allocation

The instruction sequence for cons allocation with copying collection is given in Figure A.9. Object initialization is not counted in the cost because all algorithms initialize in the same way. The following assumptions are made:

```
  # EQ test without allowing aliases:
  cmp %pointer1, %pointer2
```

```
  # EQ test with allowing aliases:
  # first, check if they are eq
  cmp %pointer1, %pointer2
  be eqtrue
  nop

  # since they are not identical check for aliases

  cmp %pointer1, %g_fromspaceBase
  blu checkPointer2
  nop
  cmp %pointer1, %g_fromspaceTop
  bgu checkPointer2
  nop

  # relocate %pointer1
  call readBarrierMove
  mov %pointer1, %o0
  mov %o1, %pointer1

checkPointer2:
  cmp %pointer2, %g_fromspaceBase
  blu done
  nop
  cmp %pointer2, %g_fromspaceTop
  bgu done
  nop

  # relocate %pointer2
  call readBarrierMove
  mov %pointer2, %o0
  mov %o1, %pointer2

done:
  cmp %pointer1, %pointer2
eqtrue:
```

Figure A.7: Modified Eq Instruction Sequence

```
  # pointer store without fromspace test:
  st %object, %container + offset
```

```
  # pointer store with fromspace test:

  # test against the bottom of fromspace
  cmp %object, %g_fromspaceBase
  blu,a done
  st %object, %container + offset

  # test against the top of fromspace.
  cmp %object, %g_fromspaceTop
  bgu,a done
  st %object, %container + offset

  # here, the pointer points into fromspace
  # call the function to relocate the object and
  #      update the pointer (the new pointer is in %o1)
  call readBarrierMove
  mov %object, %o0
  mov %o1, %dest
  st %object, %container + offset

done:
```

Figure A.8: Instructions Preventing Stores of Fromspace Pointers.

1. `%g_allocated` counts down the amount of the allocation threshold remaining to be allocated before a garbage collection.

2. `%g_copy` is the location where new objects are allocated.

3. The new cons is pointed to by `%result`.

```
  # update threshold count
  sub %g_allocated, ConsSize, %g_allocated
  bgz,a okay
  mov %g_copy, %result
  call stcpCollect
  nop
  mov %g_copy, %result
okay:
  # here, we've collected if needed
  add %g_copy, ConsSize, %g_copy
  or %result, ConsType, %result
done:
```

Figure A.9: Stop-and-Copy Allocation Sequence

The cost of an allocation using this sequence is five instructions. An even shorter sequence could be used if the allocation threshold was marked by placing an unwritable page at the proper place in the address space and changing the protection-fault handler to check for and perform the collection (as was envisioned for SPUR Lisp [96]). Using a page trap reduces the cost to three instructions per allocation but makes the collector more operating system dependent. The five instruction cost is used in the comparisons in Chapter 5.

The Scanning Inner Loop

The cost of copying collection is largely determined by the cost of the scanning inner loop, which scans copied objects, updating pointers and copying when necessary. Figure A.10 shows a simplified flow graph indicating the necessary basic operations and their cost in cycles. This figure was used to compute the cost of copying collection used in Chapter 5.

Simplifications include the grouping of all non-cons, fixed-sized objects and all variable-sized objects into two generic types. Instruction counts labeled as *variable* indicate that the cost is related to the size of the object. By counting the number of occurrences of the basic operations pictured in the flow graph, an estimate of the cost of the implementation in cycles can be determined.

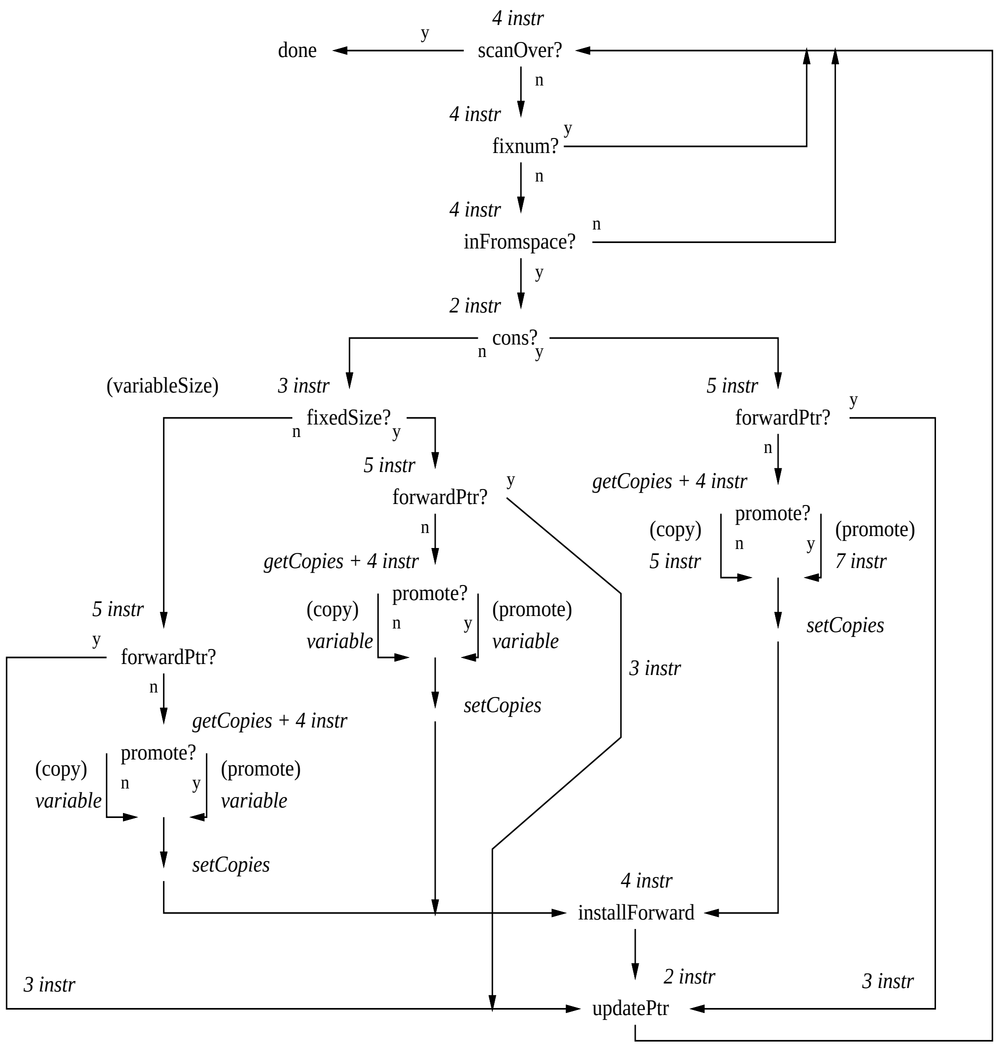

Figure A.10: Flowgraph for Computing Stop-and-Copy Execution Time

The instruction costs in the figure were computed from the SPARC assembler code sequences presented in Figures A.11 and A.12. This code sequence makes several important assumptions.

1. This inner loop handles only collections of newspace where promotion goes to the second generation. Since this is by far the most common it makes sense for the collector to handle it with a special case.

2. The register `%scan` points to the place in newspace where scanning takes place. Objects copied within newspace are copied to `%g_copy` and promoted objects are copied to `%promoted`. When `%scan` equals `%g_copy`, then the collection is finished.

3. The top and bottom of fromspace are delineated by `%fromspaceTop` and `%fromspaceBase`, respectively.

4. Fixnums have a type with value zero.

Only the instruction sequence for cons objects is provided for brevity. The code sequences for relocating other objects are similar to the sequence shown.

In the figures, two pseudo-instructions are used, `getCopies` and `setCopies`. These represent the instruction sequences that determine how many times each object has been copied. Because this sequence depends on how the copy count data is stored, it is implementation dependent. If counts are associated with objects or groups of objects, accessing the count requires a dereference and possible address arithmetic to determine the associated group. I estimate that `getCopies` will require four instructions. Once the location of the copy count is determined in `getCopies`, `setCopies` simply performs a store to the proper location, at a cost of one instruction.

A.6.2 Mark-and-Sweep Collection

Allocation Cost

In mark-and-sweep collection, allocation requires removing an element from a free list of objects of the particular type. This list may be empty, so a check must be made. After the check, the next element of the list is accessed, and the list updated. In the instruction sequence in Figure A.13, which allocates a cons object, the list is maintained as a vector and the list is empty when the vector index reaches zero. The cost of allocating a cons object is five cycles when no marking or sweeping is required. For fixed-size, non-cons objects, two additional cycles are required, because the free list index and free list vectors will not be accessible via global registers. For vector objects, eight additional cycles are required to allocated the variable-sized part of the object and link the parts together. It is clear from the sequence and discussion that mark-and-sweep allocation is slightly more expensive than copying allocation.

```
again:
  # fixnum?
  ld %scan, %ref
  and %ref, TypeMask, %type
  bz scanOver?

  # inFromspace?
  cmp %ref, %fromspaceBase
  blt scanOver?
  cmp %ref, %fromspaceTop
  bge scanOver?

  # cons?
  cmp %type, ConsType
  bne otherType
  nop

  # forwardPtr?
  ld %ref + ConsAdjust, %obj0
  and %obj0, ForwardTypeMask, %tmp
  cmp %tmp, ForwardType
  bne nonForward
  invert ForwardTypeMask, %invertmask
  and %obj0, %invertmask, %newloc
  jmpl updatePtr
  nop

nonForward:
  # decide to promote or copy
  getCopies %ref, %copies

  # promote?
  cmp %copies, CopyNumber
  blt copy
  nop

promote: # (see next figure)
```

Figure A.11: Stop-and-Copy Scanning Inner Loop (page 1)

```
promote:
  st %obj0, %promote + 0
  ld %ref + ConsAdjust, %tmp
  st %tmp, %promote + CellSize
  mov %promote, %newloc
  add %promote, ConsSize, %promote
  setCopies %ref, 0
  jmpl installForward
  nop

copy:
  st %obj0, %g_copy + 0
  ld %ref + ConsAdjust, %tmp
  st %tmp, %g_copy + CellSize
  mov %g_copy, %newloc
  add %g_copy, ConsSize, %g_copy
  setCopies %ref, %copies + 1

installForward:
  invert ForwardTypeMask, %invertmask
  and %newloc, %invertmask, %forward
  or %forward, ForwardType, %forward
  st %forward, %ref + ConsAdjust

updatePtr:
  or %newloc, ConsType, %newref
  st %newref, %scan

scanOver?:
  add %scan, CellSize, %scan
  cmp %scan, %g_copy
  bne again
  nop

done:
```

Figure A.12: Stop-and-Copy Scanning Inner Loop (page 2)

```
  # update threshold count
  sub %g_allocated, ConsSize, %g_allocated
  bgz,a noCollect
  sub %g_freeConsIndex, 4, %g_freeConsIndex
  call mdsCollect
  nop
  sub %g_freeConsIndex, 4, %g_freeConsIndex
noCollect:
  # check for need to sweep
  bgz,a done
  load %g_freeCons + %g_freeConsIndex, %result
  call deferredSweep
  nop
  load %g_freeCons + %g_freeConsIndex, %result
done:
```

Figure A.13: Mark-and-Sweep Allocation Sequence

Sweep Cost

The cost of sweeping the mark bitmap can be computed by looking at the cost of investigating a single bit. I assume that bitmaps are investigated a word at a time and all bits in the word are scanned by unrolling a loop that looks at each of the 32 bits, reducing loop overhead. A sequence of the unrolled loop that looks at a single bit in a cons bit vector is shown in Figure A.14. The cost of this sequence is four instructions if the bit is set and seven instructions if the bit is not set. Thus the cost of sweeping per object recovered (approximately seven instructions if the bitmap is sparsely marked), is close to the cost of allocation.

The Marking Inner Loop

The cost of marking is largely determined by the marking inner-loop, which continually pops references off a stack, checks the type, generation, and mark of each object, and marks the object if necessary. This flow of operations, with associated cycle costs, is illustrated in Figure A.15. When compared with the copying flow graph, the figure illustrates how similar the two algorithms really are. The instruction counts from the figure were computed from the instruction sequences in Figures A.16 and A.17. Again, the code for non-cons objects is omitted.

The largest cost in the inner loop is the testing of the mark bit in the bitmap. For the SPARC processor, this test requires 12 instructions. Fortunately, once the byte location in the bitmap is determined, setting the mark bit is an inexpensive operation (two

```
  # %bitsLeft contains the remaining bits in the bitmap word
  and %bitsLeft, 1, %thisBit
  bnz,a nextBit
  add %currentRef, ConsSize, %currentRef

  # the bit is clear, so reclaim the word by putting its
  # address in the freeVector
  st %currentRef, %g_freeCons + %g_freeConsIndex
  add %g_freeConsIndex, 4, %g_freeConsIndex
  add %currentRef, ConsSize, %currentRef

nextBit:
  # update %currentRef to reflect the bitmap position
  srl %bitsLeft, 1, %bitsLeft

# etc. for the remaining bits in the word...
```

Figure A.14: Mark-and-Sweep Sweeping Inner Loop

instructions). The code that immediately checks the cdr of a cons cell avoids the overhead of pushing and then immediately popping the cdr, a considerable savings if most objects marked are cons objects.

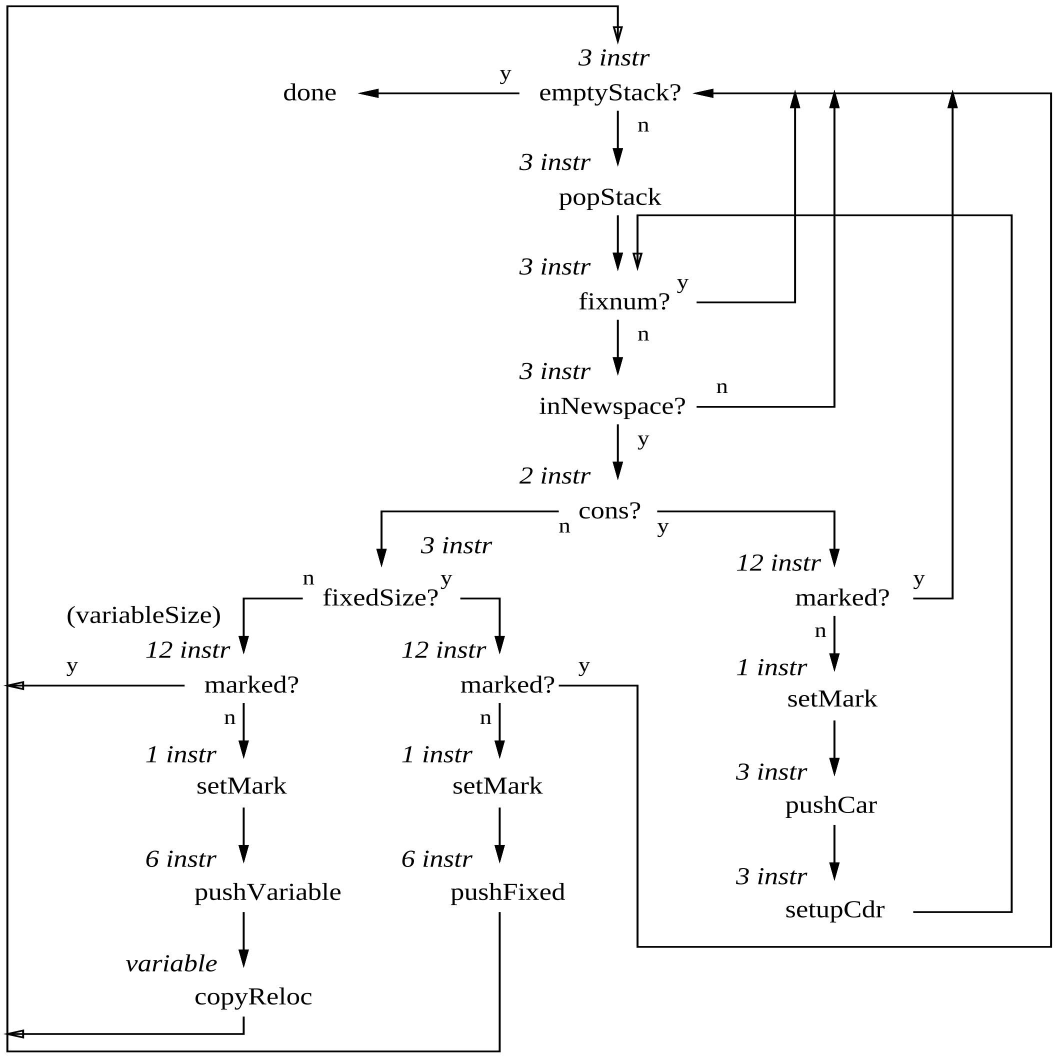

Figure A.15: Flowgraph for Computing Mark-and-Sweep Execution Time

```
emptyStack?:
  cmp %stackPointer, %stackBottom
  beq done
  nop

  # popStack
  ld %stackPointer, %loc
  ld %loc, %ref
  sub %stackPointer, 4, %stackPointer

checkCdr:
  # fixnum?
  and %ref, TypeMask, %type
  bz emptyStack?

  # inNewspace?
  cmp %ref, %fromspaceBase
  blt emptyStack?
  cmp %ref, %fromspaceTop
  bge emptyStack?

  # cons?
  cmp %type, ConsType
  bne otherType

inNewspace: (see next figure)
```

Figure A.16: Mark-and-Sweep Marking Inner Loop (page 1)

```
inNewspace:
  # marked?
  # get area and bitmap from ref by shifting and masking upper bits
  sll %ref, LogAreaSize, %areaIndex
  and %areaIndex, AreaMask, %areaIndex
  ld %newspaceAreas + %areaIndex, %areaPointer
  ld %areaPointer + BitmapOffset, %bitmapBase

  # calculate the bit and byte offset in the bitmap
  and %ref, AreaBitMask, %tmp
  sll %tmp, 3, %bitoffset
  sll %bitoffset, 3, %byteoffset

  # create a mask with a 1 set in the correct place of a byte
  and %bitoffset, 3, %bitinbyte
  srl 1, %bitinbyte, %bitmask

  # get the byte and test the bit
  ldb %bitmapBase + %byteoffset, %byte
  and %byte, %bitmask, %tmp
  bnz emptyStack?

  # setMark
  or %byte, %mask, %byte
  stb %byte, %bitmapBase + %byteoffset

  # pushCar
  ld %ref + ConsAdjust, %tmp
  st %tmp, %stackPointer
  add %stackPointer, 4, %stackPointer

  # setupCdr
  add %ref, ConsAdjust, %loc
  jmpl checkCdr
  nop
done:
```

Figure A.17: Mark-and-Sweep Marking Inner Loop (page 2)

Appendix B

Formal Algorithm Definitions

This appendix contains formal definitions of the algorithms compared in the body of the thesis. The naming conventions are as follows. Variables that are global to the computation begin with a capital letter (e.g., *Memory*). Variables local to a function and function names begin with a lower-case letter (e.g., *result*). All variables and function names are printed in italic font (e.g., *New*). Constants are printed in san-serif font (e.g., $\mathsf{CopyNumber}$). In this section, the symbol "$\equiv$" means "is defined as."

A *generation number*, g, is the index of a generation, ranging from the youngest generation ($gNew$), with index 0, to the oldest generation ($gOld$) with some small integer index. An *address* (or offset within a generation), a, is a non-negative integer. A *location* is defined to be a pair of generation number and address, $l = \{g, a\}$. Two operations, *address* and *gen* are defined on locations. If $l = \{g, a\}$ is a location, $address(l) \equiv a$, and $gen(l) \equiv g$.

A *type* is any Lisp type. In general the set of all Lisp types, $\mathsf{AllTypes}$, is implementation specific, but must contain the following types: Fixnum, Cons, and $\mathsf{Forward}$, which is the type given to forwarding pointers.

A *reference* is a type, location pair. If $r = \{l, t\}$, is a reference, then $type(r) \equiv t$ and $location(r) \equiv l$. Furthermore, *address* and *gen* are polymorphically defined on references so that $address(r) \equiv address(location(r))$ and $gen(r) \equiv gen(location(r))$. The function *makeReference* takes a location and a type and creates a reference.

A *memory cell* contains either a reference or binary data. The global variable *Memory* is an two-dimensional array of memory cells indexed by generation number and location. As a shorthand, if l is a location, $Memory[l] \equiv Memory[gen(l)][address(l)]$. A predicate, *boxed?*, when applied to a location, can be used to determine if the memory cell at that location contains a reference or binary data.

A reference is either an immediate or refers to an associated object. The predicate $immediate?(r) \equiv (type(r) = \mathsf{Fixnum})$, is used to determine if an object is an immediate. If a reference, r, is not an immediate, then the function $object(r)$ returns the associated object. If r is a reference to an object, o, then $location(o) \equiv location(r)$. Furthermore, the contents of an object at a particular offset, i, can be obtained using the following definition: $contents(o, i) \equiv Memory[location(o) + i]$. Finally, objects have an associated non-negative size, accessed using $size(o)$.

B.1 Copying Algorithms

In addition to the standard definitions, the stop-and-copy and incremental collection algorithms have other variables defined. To implement the allocation threshold, a variable (*Allocated*) and a constant (Threshold) are associated with each generation (denoted $Allocated_g$, where g is the generation). Each generation also has associated variables defining the semispaces. *ToBase*, *ToTop*, *FromBase*, and *FromTop*, are addresses that define the extents of the semispaces for each generation. *Scan* and *Copy* are variables used for allocation and collection in each generation. New objects in generation g are allocated at the location $Copy_g$. During a collection, the location $Scan_g$ indicates where scanning is taking place.

B.1.1 Stop-and-Copy Collection

Figures B.1, B.2, and B.3 show the pseudocode needed to define a simple generation-based stop-and-copy collection algorithm. In this description, the symbols "$+\leftarrow$" and "$-\leftarrow$" are used to indicate the increment and decrement operations, respectively, in a notation borrowed from C. Pseudocode subroutines are one of three types: **function**, which returns a value; **procedure** which is invoked for effect and does not return a value; and **predicate**, which returns a boolean value.

The only routine made external by the stop-and-copy collection algorithm is *stcpAllocate*, which is called to allocate a new object (shown in Figure B.1). That function simply checks to see if the allocation threshold has been exceeded. If so a collection is initiated, otherwise memory is allocated at the location $Copy_{gNew}$ and a reference to the new object is created and returned.

When a collection is required, the sequence of events is simple. First, the generations that need collection are determined. This implementation of generation collection requires that if generation g needs collection, g and all generations younger must be collected (see *pickGenerations*). Then, the semispaces in these generations must be flipped (*flipSemispaces*). Next, the root set must be scanned and transported if necessary (*scanRootSet*) , and finally the objects copied to tospace must be scanned and reachable objects transitively transported (*scanLoop*).

The procedure *transportAndUpdate* takes a location and determines if the reference at that location refers to an object that needs to be transported or has been transported. It first checks for a forwarding pointer, in which case it recovers the location needed to update the current reference. Next it checks if the object needs to be transported. If it does, the object is transported and the reference to it is updated. The predicate *forward?* simply checks for a pointer installed with the type Forward. The predicate *transport?* returns True

```
function stcpAllocate(type, size) is
    if (size + Allocated_gNew) > Threshold_gNew then
        stcpCollect()
    endif
    result ← Copy_gNew
    Copy_gNew +← size
    Allocated_gNew +← size
    return makeReference(result, type)
end
```

```
procedure stcpCollect() is
    CollectGens ← pickGenerations()
    flipSemispaces()
    scanRootSet()
    scanLoop()
end
```

```
function pickGenerations() is
    for g from gOld to gNew do
        if Allocated_g > Threshold_g then
            return {0, ..., g}
        endif
    endfor
end
```

```
procedure flipSemispaces() is
    for g ∈ CollectGens do
        swap(ToTop_g, FromTop_g)
        swap(ToBase_g, FromBase_g)
        Scan_g ← ToBase_g
        Copy_g ← ToBase_g
        Allocated_g ← 0
    endfor
end
```

Figure B.1: Stop-and-Copy Pseudocode (page 1)

```
procedure scanRootSet() is
    for l ∈ RootSet do transportAndUpdate(l)
end
```

```
procedure scanLoop() is
    allScanned? ← False
    while not allScanned? do
        allScanned? ← True
        for g ∈ {gNew, ..., gOld } do
            while Scan_g < Copy_g do
                allScanned? ← False
                if boxed?(Scan_g) then
                    transportAndUpdate(Scan_g)
                endif
                Scan_g +← 1
            endwhile
        endfor
    endwhile
end
```

```
procedure transportAndUpdate(l) is
    r ← Memory[l]
    if forward?(r) then
        newloc ← location(contents(r,0))
    else if transport?(r) then
        newloc ← transport(r)
    else
        return
    endif
    newref ← makeReference(newloc, type(r))
    Memory[l] ← newref
end
```

```
predicate forward?(r) is
    if immediate?(r) return False
    else return type(contents(r,0)) = Forward
end
```

Figure B.2: Stop-and-Copy Pseudocode (page 2)

if the reference is not an immediate and points to an object located in the fromspace of a generation being collected.

The function *transport* first determines what generation the object should be transported to using the function *promoteDecision*, which uses a copy count promotion strategy. *Transport* then updates the allocation count in the assigned generation and allocates memory for the copy. Each word of the object is copied, and then a forwarding pointer is installed, indicating where the object has been copied. Finally, the copy count of the copied object is either zeroed, when the object is promoted, or incremented, when it is simply transported.

```
predicate transport?(r) is
    if immediate?(r) then
        return False
    else
        g ← gen(r)
        return g ∈ CollectGens and location(r) ∉ [ToBase_g, ToTop_g]
    endif
end
```

```
function transport(r) is
    toGen ← promoteDecision(r)
    o ← object(r)
    if toGen > gNew then
        Allocated_toGen +← size(o)
    endif
    newloc ← Copy_toGen
    for i ∈ [0, ..., size(o) − 1] do
        Memory[Copy_toGen] ← contents(o, i)
        Copy_toGen +← 1
    endfor
    forward ← makeReference(newloc, Forward)
    contents(o, 0) ← forward
    if toGen > gen(r) then
        copyCount(o) ← 0
    else
        copyCount(o) +← 1
    endif
    return newloc
end
```

```
function promoteDecision(r) is
    if copyCount(object(r)) ≥ CopyNumber then
        return min(gen(r) + 1, gOld)
    else
        return gen(r)
    endif
end
```

Figure B.3: Stop-and-Copy Pseudocode (page 3)

B.1.2 Incremental Collection

Because incremental collection is a variant of stop-and-copy collection, many of the functions defined for the previous algorithm can be reused. The major difference between the algorithms lies in their behavior during allocation. Figures B.4 and B.5 contain pseudocode for the incremental variation of a generation stop-and-copy algorithm. The entry point is the function *incrAllocate*, which allocates an object of the require type and size and returns a reference to it. Before allocating the new object, a fixed number of memory cells in tospace are scanned as part of the incremental collection. A global variable, *AllScanned?*, indicates whether further scanning is necessary. Incremental collection allocates new objects in tospace at a different location than where objects are copied because new objects do not have to be scanned as copied objects do. New objects are allocated from the top of the semispace growing downward. Copied objects are allocated at the bottom of the semispace growing upward.

When scanning is required, *scanKCells* is called. This routine scans exactly ScansPerAlloc cells and then returns. The loop is similar to *scanLoop* in the stop-and-copy algorithm, except that a count of the number of cells scanned is also maintained. When *Scan* equals *Copy* in all generations, then no further scanning is necessary.

When the incremental algorithm flips, everything is done just like the stop-and-copy algorithm, except that scanning of transported objects is deferred and done incrementally. The major overhead of incremental collection is the cost of loading a pointer (function *incrLoadPointer*), because tests must be made to guarantee the pointer does not refer to fromspace. Fast implementations of this test are discussed in Chapter 5.

```
function incrAllocate(type, size) is
    if not AllScanned? then
        scanKCells()
    endif
    if (size + Allocated_gNew) > Threshold_gNew then
        incrFlip()
    endif
    New −← size
    Allocated_gNew +← size
    return makeReference(New, type)
end
```

```
procedure scanKCells() is
    scanCount ← 0
    while not AllScanned? do
        AllScanned? ← True
        for g ∈ {gNew, ..., gOld} do
            if Scan_g = Copy_g then
                continue
            else
                AllScanned? ← False
                while Scan_g < Copy_g and scanCount < ScansPerAlloc do
                    if boxed?(Scan_g) then
                        transportAndUpdate(Scan_g)
                    endif
                    Scan_g +← 1
                endwhile
                if scanCount = ScansPerAlloc then
                    return
                endif
            endif
        endfor
    endwhile
end
```

Figure B.4: Incremental Pseudocode (page 1)

```
procedure incrFlip() is
    CollectGens ← pickGenerations()
    flipSemispaces()
    New ← ToTop_gNew
    scanRootSet()
    AllScanned? ← False
end
```

```
function incrLoadPointer(r) is
    if transport?(r) then
        newloc ← transport(r)
        return makeReference(newloc, type(r))
    endif
end
```

Figure B.5: Incremental Pseudocode (page 2)

B.2 Mark-and-Deferred-Sweep Collection

This variation of a generation mark-and-sweep algorithm requires additional definitions, functions, and variables. First, a special representation of objects introduces additional functions and constants. The set of all types, AllTypes is partitioned into types with a fixed size (e.g., Cons) and types whose objects can have different sizes (e.g., Vector). The predicate *fixedSize?* takes a type and returns a boolean indicating whether objects of the type have a fixed size.

Objects with a type of a variable size are divided into two parts, the *header* and the *body*. The location of the body of a variable-sized object is stored at a constant offset (LinkOffset) in the header. The size of the body is also stored at a constant offset (SizeOffset) in the header. A reference to a variable-sized object contains the location of the header.

In addition to the new representation of objects, the mark-and-deferred-sweep algorithm requires a number of additional global variables. Each generation is divided into two parts: a part that contains fixed-sized objects and object headers, and a part that contains object bodies. All references refer to the memory in the region containing fixed-size objects. The memory that contains object bodies is split into two semispaces just like a copying algorithm. The variables *FromBase*, *FromTop*, *ToBase*, *ToTop*, and *Copy* all refer to the memory used to contain vector bodies.

The global variable *FreeSet(type, gen)* maps a type and a generation number to a set references to free objects of that type in that generation. Defined set operations include *emptySet?*, *pushSet*, and *popSet*. The memory for fixed-sized objects in each generation is further subdivided into smaller units and each unit is assigned a specific type. These regions, called *areas*, have an associated type (*areaType*) and bitmap (*areaBitmap*), which is used to mark objects in the area. The global variable *AreaSet(type, gen)*, contains the set of areas in a particular generation of a particular type. Areas are allocated from an infinite supply of free areas accessed with variable *FreeAreaSet(gen)*.

The bits in a bitmap correspond to locations in the memory. The function *bitmapOffsetToLocation(bitmap, offset)* returns the location that corresponds to an offset in an area bitmap. The inverse function, *locationToBitmapOffset(l)*, maps a location to an offset in an area bitmap. Other operations on bitmap b at offset i include the predicate *bitSet?(b, i)*, which indicates if an entry is set, *setBit(b, i)*, which sets a bit, and *clearBitmap(b)*, which clears an entire area bitmap.

A final data structure, the *MarkStack*, is used to stack locations of cells yet to be examined. Stack operations *emptyStack?*, *popStack*, and *pushStack* are defined.

With this introduction, the pseudocode can now be described. The entry point is *mdsAllocate*, which allocates an object given a type and size. The size is only necessary for variable-sized objects and indicates the size of the body. *mdsAllocate* checks the collection threshold and then allocates the object. Fixed-sized objects are allocated in *allocateFixed*. Variable-sized objects have their headers allocated in *allocateFixed* and their bodies allocated in *allocateBody*, after which *mdsAllocate* sets up the size and link information and

```
function mdsAllocate(type, size) is
    if (size + Allocated_gNew) > Threshold_gNew then
        mdsCollect()
    endif
    Allocated_gNew +← size
    if (fixedSize?(type) then
        return allocateFixed(gNew, type)
    else
        bodyLoc ← allocateBody(gNew, size)
        header ← allocateFixed(gNew, type)
        contents(header, LinkOffset) ← bodyLoc
        contents(header, SizeOffset) ← size
        return header
    endif
end
```

```
function allocateFixed(gen, type) is
    if not emptySet?(FreeSet(gen, type)) then
        result ← popSet(FreeSet(gen, type))
        mark(result)
        return result
    else
        deferredSweep(gen, type)
        return allocateFixed(gen, type)
    endif
end
```

```
function allocateBody(gen, size) is
    result ← Copy_gen
    Copy_gen +← size
    return result
end
```

Figure B.6: Mark-and-Deferred-Sweep Pseudocode (page 1)

returns a reference to the header.

The function *allocateBody* simply allocates the body in the memory region reserved for vector bodies and returns the location. *allocateFixed* attempts to remove an element from the *FreeSet* associated with the type and generation, and returns the reference if the *FreeSet* is not empty. Otherwise, it performs a deferred sweep of the bitmaps associated with areas of the proper type and then returns the reference. The function *deferredSweep* sweeps

```
procedure deferredSweep(gen, type) is
    sweepCount ← 0
    for a ∈ AreaSet(gen, type) do
        for b ∈ bitmap(a) do
            if not bitSet?(b) then
                sweepCount +← 1
                newref ← makeReference(bitmapOffsetToLocation(offset(b)), type)
                pushSet(newref, FreeSet(gen, type))
                if sweepCount = SweepsPerAlloc then
                    return
                endif
            endif
        endfor
    endfor
    pushSet(allocateArea(gen, type), AreaSet(gen, type))
    deferredSweep(gen, type)
end
```

```
function allocateArea(gen, type) is
    newArea ← popSet(FreeAreaSet(gen))
    areaType(newArea) ← type
    return newArea
end
```

Figure B.7: Mark-and-Deferred-Sweep Pseudocode (page 2)

the bitmaps associated with areas allocated to a particular type and generation, creating references when it finds locations that are not marked. The constant SweepsPerAlloc controls how often deferred sweeping takes place. If not enough objects are swept in all the areas allocated, a new area of the proper type is allocated with *allocateArea* and the new area is then swept.

The function *mdsCollect* performs the complex mark-and-deferred-sweep collection. The generations needing collection are placed in *CollectGens*. Next, what kind of promotion is necessary is decided. The function *pickPromoteGen* determines what the destination generation for promotion should be. The variable *collectCount* records the number of collections that have been performed in each generation since the last promotion. The constant CollectCountThreshold determines how many collections before promotion. The oldest generation, g, where $CollectCount_g \geq CollectCountThreshold_g$ is oldest generation needing promotion. All objects in that generation and younger ones are promoted to $g + 1$ (en masse promotion).

Depending on whether promotion is needed at all, and which generation is the destination of promotion, either marking, transporting, or both are done. If no promotion is necessary, then just marking is performed (by *markPhase*). If *PromoteGen* is bigger than all the generations needing collection, then no marking is necessary because all collected objects are promoted (using *transportPhase*). Otherwise, first objects need to be marked, and then some of the generations need promotion (both phases).

```
procedure mdsCollect() is
    CollectGens ← pickGenerations()
    PromoteGen ← pickPromoteGen()
    if PromoteGen = {} then
        markPhase()
    else if PromoteGen = (max(CollectGens) + 1) then
        transportPhase()
    else
        markPhase()
        transportPhase()
    endif
end
```

```
function pickPromoteGen() is
    for g from gOld to gNew do
        if CollectCount_g ≥ CollectCountThreshold_g then
            return max(gOld, g + 1)
        endif
    endfor
    return {}
end
```

Figure B.8: Mark-and-Deferred-Sweep Pseudocode (page 3)

A mark phase marks all objects in the appropriate generations and copies and compacts the relocatable bodies of vectors. First, the semispaces (for vector bodies) are flipped, as in copying collection. Then the appropriate bitmaps are cleared. Finally, the root set is marked and the mark stack is transitively examined. The inner loop of the algorithm pops locations from *MarkStack*, dereferences them to obtain a reference, checks if the referenced object has been marked, marks it if necessary, relocates bodies of vectors, and then pushes all locations in the marked object onto the stack. This loop continues until the *MarkStack* is empty.

An object does not need to be marked if it is an immediate, is not in the proper generation, or has already been marked. An object is marked by setting a bit in the bitmap which maps the area containing the object. A mark is tested by checking that bit. Finally, relocation the body of a vector simply requires copying the contents of the body. In this algorithm, references do not have to be relocated until the transport phase because all references are to object headers or to fixed-size objects, which are not relocated until they are promoted. The transport phase of this algorithm is not presented because it is similar to the marking phase, except that instead of simply marking objects, they are copied and forwarding pointers are left behind.

```
procedure markPhase() is
    for g ∈ CollectGens do
        swap(ToBase_g, FromBase_g)
        swap(ToTop_g, FromTop_g)
        Copy_g ← ToBase_g
    endfor
    for g ∈ CollectGens do
        for t in AllTypes do
            for a ∈ AreaSet(g, t) do
                clearBitmap(bitmap(a))
            endfor
        endfor
    endfor
    for l ∈ RootSet do
        pushStack(l, MarkStack)
    endfor
    markFromStack()
end
```

```
procedure markFromStack() is
    while not emptyStack?(MarkStack) do
        l ← popStack(MarkStack)
        if not boxed?(l) then
            continue
        endif
        r ← Memory[l]
        if not needsMarking?(r) then
            continue
        endif
        mark(r)
        if not fixedSize?(type(r)) then
            relocateBody(r)
        endif
        for l ∈ locations of references in r do
            pushStack(l, MarkStack)
        endfor
    endwhile
end
```

Figure B.9: Mark-and-Deferred-Sweep Pseudocode (page 4)

```
predicate needsMarking?(r) is
    return not (immediate?(r) or gen(r) ∉ CollectGens or marked?(r))
end
```

```
procedure mark() is
    a ← area containing location of r
    b ← bitmap(a)
    setBit(b, locationToBitmapOffset(location(r)))
end
```

```
predicate marked?(r) is
    a ← area containing location of r
    b ← bitmap(a)
    return bitSet?(b, locationToBitmapOffset(location(r)))
end
```

```
procedure relocateBody(r) is
    bodySize ← contents(r, SizeOffset)
    bodyLoc ← contents(r, LinkOffset)
    g ← gen(r)
    newloc ← Copy_g
    for i ∈ [0, ..., bodySize − 1] do
        Memory[Copy_g] ← Memory[bodyLoc + i]
        Copy_g +← 1
    endfor
end
```

Figure B.10: Mark-and-Deferred-Sweep Pseudocode (page 5)

Appendix C

Four More Programs

This appendix presents the results measured in the thesis for four additional test programs, including the data in a tabular form. Table C.1 summarizes the four programs. As the

Resource Description	RSIM	PC	Weaver	PMA
Source lines	2800	4700	7200	5100
Execution time (sec), w/o monitor	294	92.6	94.7	90.6
Execution time (sec), w. monitor	3889	3446	2045	1586
Monitor factor slowdown	13	37	21	17
Program references (millions)	37.6	41.9	29.1	20.1
Objects allocated (millions)	6.9	3.9	0.08	1.1
Bytes allocated (millions)	57.2	35.2	0.69	14.8

Table C.1: General Information about Additional Test Programs.

table indicates, the additional programs are smaller, run for fewer references, and allocate less data than the programs examined in the body of the thesis. One of the programs, Weaver, an OPS5 application, allocates very little data. These programs are still larger and more interestingly behaved than simple benchmarks, like those reported by Gabriel. This appendix briefly describes the programs and then presents the measurements reported in the body of the thesis as applied to these programs.

C.1 RSIM

RSIM is transistor-level circuit simulator described by Terman [82]. The simulator is measured simulating a simple 10-bit binary counter counting to 1000. RSIM is unusual in that it allocates many floating point numbers—far more than any of the other test programs. Unfortunately, RSIM behaves very regularly on this input because in simulating a counter, each count requires the same allocation and reference behavior. RSIM is similar to a benchmark program in which a small core is executed 1000 times. Also, RSIM is not a particularly interesting program for garbage collection studies because most of the floating point objects it allocates are very short-lived, and so little garbage accumulates during its execution. However, since it has been used in other studies (e.g., Bob Shaw's thesis [74]), RSIM is a good test program.

RSIM uses a relatively modern programming style, including structures. Most of the structure objects are used to represent the circuit, and hence are allocated initially and survive the duration of the program. The majority of the 60 megabytes of objects allocated throughout its execution are floating point numbers and cons cells.

C.2 A Prolog Compiler (PC)

The Prolog compiler translates Warren abstract machine (WAM) instructions into SPUR instructions, performing optimization. The optimization phase accounts for the largest part of the execution. Optimization is performed by pattern matching subtrees of a tree representing the program. The pattern matching is iterative and continues until no matches are found, which for the test input takes three iterations. The test input is WAM for a small Prolog program the calculates the Sieve of Eratosthenes. The Prolog compiler is very memory intensive, and allocates many cons cells during the matching phase of optimization. Part of the reason for the intense allocation is the program's author felt that memory allocation should be cheap and made no attempt to reduce the memory requirements of the compiler.

C.3 Weaver

Weaver is a network routing program written in OPS5. The OPS5 interpreter was originally written in Maclisp. The interpreter accounts for about 3700 lines of input with the OPS5 program representing the router requiring another 3500 lines of code. This OPS5 interpreter is carefully written and painstakingly avoids allocation. The result is that the overhead of garbage collection in this program is minimal in many cases. For large enough newspace thresholds, garbage collection does not even take place. Still, Weaver is a significant program representing a common use for Lisp—prototyping new languages.

C.4 The Perq Microcode Assembler (PMA)

This microcode assembler takes microcode descriptions of the Perq architecture and generates a down-loadable microcode image. The assembler files describe the microcode implementation of Spice Lisp. The assembler reads in the microcode and generates the executable image after making several passes over the input. Both structures and cons cells are heavily used in this program.

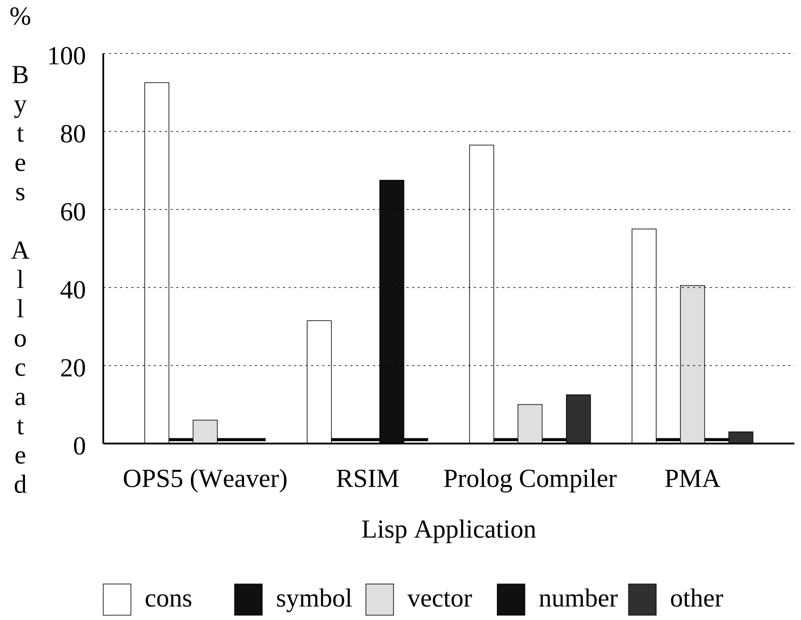

Figure C.1: Object Allocations for Test Programs (by type and size).

Lisp Application	Fraction of Bytes Allocated (%)				
	cons	symbol	vector	number	other
OPS5 (Weaver)	92.90	0.07	6.42	0.01	0.59
RSIM	31.55	0.00	0.74	67.70	0.01
Prolog Compiler	76.98	0.00	10.30	0.00	12.72
PMA	55.23	1.13	40.58	0.00	3.06

Table C.2: Object Allocations for Test Programs (by type and size).

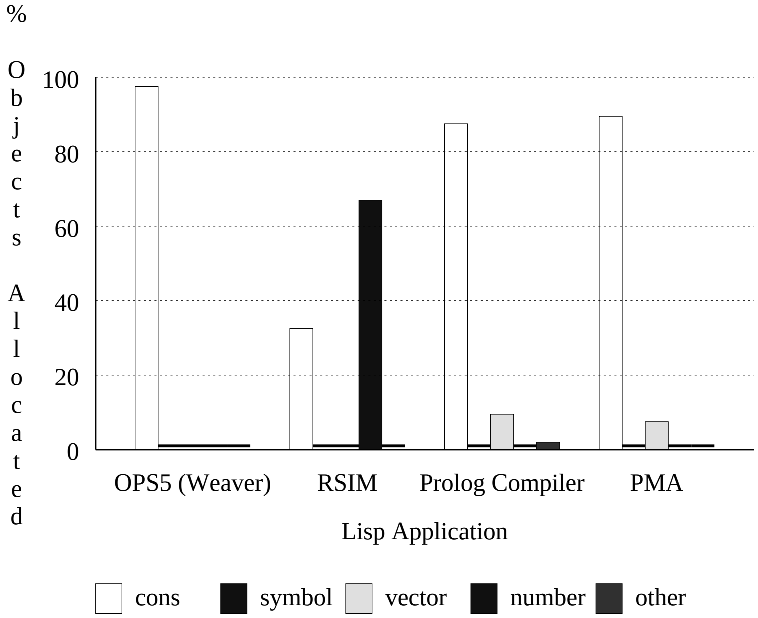

Figure C.2: Object Allocations for Test Programs (by type and number).

Lisp Application	Fraction of Objects Allocated (%)				
	cons	symbol	vector	number	other
OPS5 (Weaver)	97.96	0.03	1.75	0.01	0.25
RSIM	32.68	0.00	0.31	67.01	0.00
Prolog Compiler	87.56	0.00	10.00	0.00	2.44
PMA	89.90	0.62	7.88	0.00	1.61

Table C.3: Object Allocations for Test Programs (by type and number).

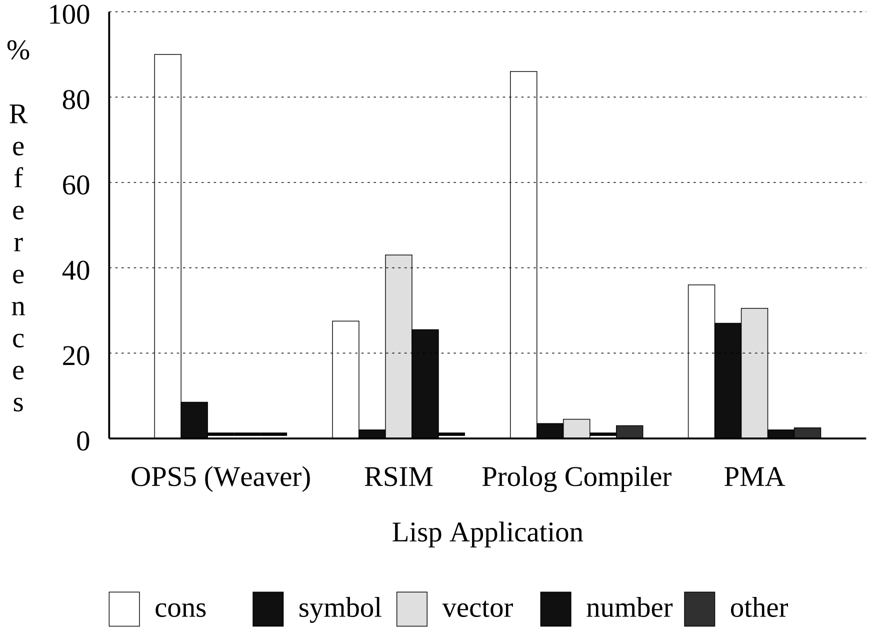

Figure C.3: Object References by Type for Test Programs.

Lisp Application	Fraction of References (%)				
	cons	symbol	vector	number	other
OPS5 (Weaver)	90.19	8.86	0.08	0.00	0.03
RSIM	27.96	2.32	43.00	25.71	0.00
Prolog Compiler	86.21	3.65	4.88	0.00	3.02
PMA	36.41	27.06	30.97	2.04	2.87

Table C.4: Object References by Type for Test Programs.

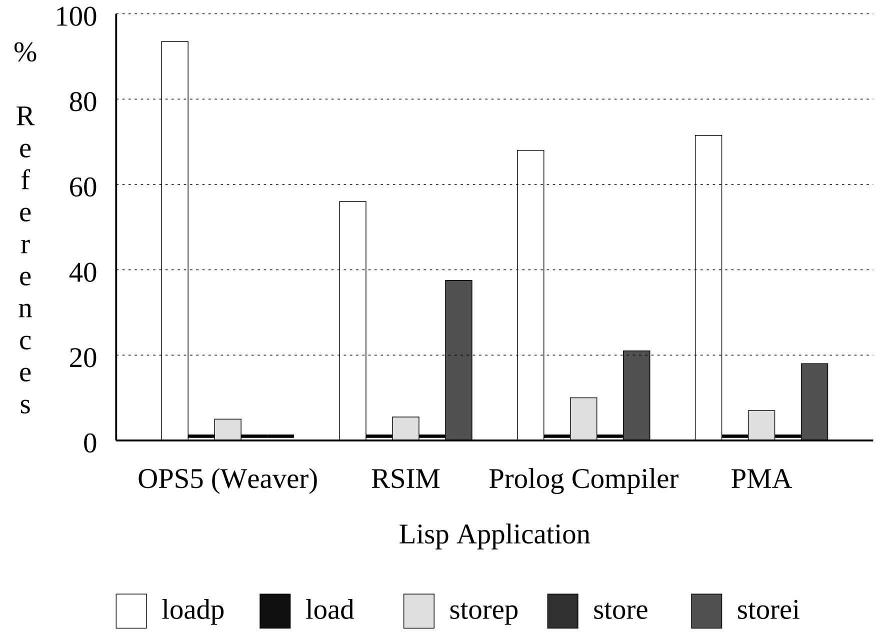

Figure C.4: Object References by Instruction Type for Test Programs.

Lisp Application	Fraction of References (%)				
	loadp	load	storep	store	storei
OPS5 (Weaver)	93.94	0.02	5.44	0.01	0.59
RSIM	56.34	0.00	5.68	0.00	37.97
Prolog Compiler	68.23	0.31	10.43	0.04	21.00
PMA	71.99	1.14	7.37	1.17	18.33

Table C.5: Object References by Instruction Type for Test Programs.

RSIM

Time (sec)	Allocation Rate (kbytes/sec)			
	cons	vector	number	other
2.5	302.1	22.4	425.4	1.3
5.0	233.9	7.1	502.7	0.0
7.5	231.0	5.7	503.2	0.0
10.0	231.8	4.8	505.3	0.0
12.5	232.1	5.0	508.0	0.0
15.1	231.7	4.8	504.8	0.0
17.6	230.8	4.9	504.0	0.0
20.1	231.3	4.6	507.8	0.0
22.6	231.8	4.8	506.1	0.0
25.1	231.3	5.0	504.8	0.0
27.6	231.9	4.8	504.5	0.0
30.1	232.0	5.0	506.6	0.0
32.6	231.8	4.8	505.6	0.0
35.1	231.5	4.8	503.5	0.0
37.6	232.2	5.0	503.2	0.0
40.1	230.8	4.6	506.5	0.0
42.7	231.2	4.7	507.1	0.0
45.2	231.7	5.0	504.0	0.0
47.7	231.6	4.8	504.6	0.0
50.2	232.0	5.0	506.5	0.0
52.7	231.8	4.8	504.6	0.0
55.2	231.9	4.8	505.0	0.0
57.7	232.1	5.0	502.6	0.0
60.2	230.6	4.7	505.7	0.0
62.7	231.8	4.7	504.8	0.0
65.2	231.6	4.9	504.9	0.0
67.8	231.5	4.7	504.5	0.0
70.3	232.2	5.0	504.5	0.0
72.8	232.0	4.8	503.7	0.0
75.3	230.9	4.8	498.0	0.0

Prolog Compiler

Time (sec)	Allocation Rate (kbytes/sec)			
	cons	vector	number	other
2.7	247.1	76.8	0.0	228.8
5.4	293.8	175.4	0.0	51.5
8.1	365.8	40.8	0.0	55.1
10.8	347.7	45.3	0.0	50.8
13.5	382.7	37.1	0.0	51.2
16.2	367.4	40.6	0.0	51.7
18.9	364.3	35.5	0.0	56.0
21.6	331.1	41.5	0.0	54.4
24.3	360.7	42.3	0.0	50.0
27.0	346.8	44.2	0.0	51.6
29.8	246.1	50.8	0.0	29.1
32.5	362.5	41.5	0.0	54.6
35.2	349.6	45.3	0.0	50.2
37.9	373.0	38.3	0.0	51.7
40.6	373.7	37.7	0.0	54.1
43.3	353.2	37.0	0.0	55.5
46.0	331.6	41.7	0.0	54.7
48.7	358.1	42.7	0.0	49.6
51.4	351.3	43.5	0.0	52.3
54.1	287.1	30.7	0.0	38.3
56.8	344.4	45.0	0.0	54.1
59.5	366.3	41.3	0.0	48.9
62.2	356.7	41.2	0.0	53.4
64.9	374.4	32.9	0.0	58.0
67.6	347.8	41.6	0.0	51.0
70.3	345.3	41.9	0.0	53.0
73.0	343.1	43.3	0.0	51.3
75.7	259.5	29.7	0.0	34.1
78.4	141.4	2.5	0.0	12.8
81.1	116.5	1.5	0.0	10.3

Table C.6: Object Allocation Rates as a Function of Time.

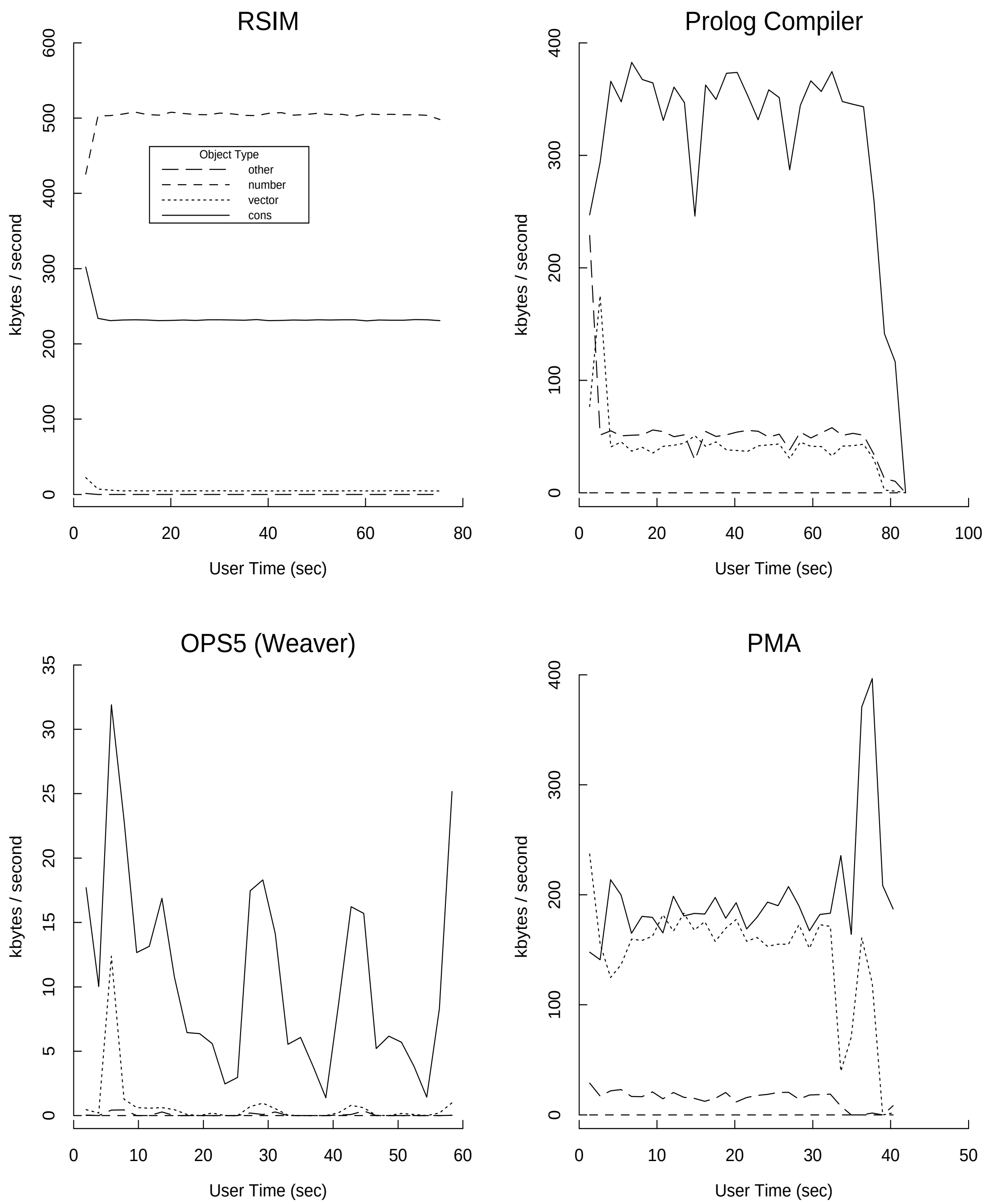

Figure C.5: Program Allocation Rates as a Function of Time.

OPS5 (Weaver)

Time	Allocation Rate (kbytes/sec)			
(sec)	cons	vector	number	other
1.9	17.7	0.5	0.0	0.0
3.9	10.0	0.2	0.0	0.0
5.8	31.9	12.4	0.0	0.4
7.8	23.0	1.3	0.0	0.4
9.7	12.7	0.6	0.0	0.0
11.7	13.2	0.6	0.0	0.0
13.6	16.9	0.6	0.0	0.3
15.5	10.8	0.5	0.0	0.0
17.5	6.4	0.1	0.0	0.0
19.4	6.4	0.0	0.0	0.0
21.4	5.6	0.2	0.0	0.0
23.3	2.5	0.0	0.0	0.0
25.3	3.0	0.0	0.0	0.0
27.2	17.5	0.7	0.0	0.2
29.2	18.3	1.0	0.0	0.1
31.1	14.1	0.5	0.0	0.3
33.0	5.5	0.0	0.0	0.0
35.0	6.1	0.0	0.0	0.0
36.9	3.8	0.0	0.0	0.0
38.9	1.4	0.0	0.0	0.0
40.8	8.6	0.2	0.0	0.0
42.8	16.2	0.8	0.0	0.1
44.7	15.7	0.6	0.0	0.4
46.6	5.2	0.0	0.0	0.0
48.6	6.2	0.0	0.0	0.0
50.5	5.7	0.2	0.0	0.0
52.5	3.8	0.1	0.0	0.0
54.4	1.4	0.0	0.0	0.0
56.4	8.3	0.2	0.0	0.0
58.3	25.2	1.0	0.0	0.0

PMA

Time	Allocation Rate (kbytes/sec)			
(sec)	cons	vector	number	other
1.3	147.9	237.2	0.0	28.9
2.7	141.0	155.5	0.0	17.1
4.0	213.8	124.8	0.0	21.8
5.4	199.8	136.5	0.0	23.0
6.7	164.9	159.6	0.0	16.7
8.1	180.3	158.5	0.0	16.5
9.4	179.4	162.3	0.0	20.8
10.8	165.4	182.0	0.0	14.6
12.1	198.7	166.9	0.0	20.2
13.4	180.8	183.2	0.0	15.9
14.8	183.0	167.9	0.0	15.0
16.1	182.4	175.6	0.0	12.4
17.5	197.5	157.6	0.0	15.0
18.8	178.6	169.6	0.0	20.4
20.2	192.8	177.5	0.0	11.7
21.5	169.1	157.8	0.0	15.7
22.9	179.7	161.2	0.0	17.7
24.2	193.4	153.1	0.0	18.6
25.5	190.1	155.1	0.0	20.4
26.9	207.5	155.1	0.0	20.5
28.2	189.9	172.8	0.0	14.2
29.6	167.2	151.3	0.0	18.0
30.9	182.2	172.7	0.0	18.4
32.3	183.2	171.3	0.0	18.7
33.6	235.7	39.5	0.0	7.9
34.9	164.0	70.8	0.0	0.0
36.3	370.9	161.0	0.0	0.0
37.6	396.4	119.8	0.0	1.6
39.0	208.4	0.0	0.0	0.0
40.3	187.0	1.8	0.0	8.3

Table C.7: Object Allocation Rates as a Function of Time.

RSIM

Time (sec)	Net Rate (kbytes/sec)			
	cons	vector	number	other
2.5	4.0	14.8	3.1	0.0
5.0	0.0	0.0	-0.1	0.0
7.5	0.0	0.0	-0.1	0.0
10.0	-2.1	-0.7	-2.2	0.0
12.5	-0.0	0.0	-0.5	0.0
15.1	0.0	0.0	-0.0	0.0
17.6	-0.0	0.0	0.1	0.0
20.1	0.0	0.0	0.2	0.0
22.6	0.0	0.0	0.0	0.0
25.1	0.0	0.0	-0.1	0.0
27.6	0.0	0.0	0.0	0.0
30.1	0.0	0.0	-0.1	0.0
32.6	0.0	0.0	0.0	0.0
35.1	0.0	0.0	0.1	0.0
37.6	-0.0	0.0	-0.2	0.0
40.1	-0.1	-0.5	0.1	0.0
42.7	0.0	0.0	0.3	0.0
45.2	0.0	0.0	-0.2	0.0
47.7	0.0	0.0	0.0	0.0
50.2	-0.0	0.0	-0.2	0.0
52.7	0.0	0.0	0.1	0.0
55.2	0.0	0.0	0.1	0.0
57.7	0.0	0.0	-0.3	0.0
60.2	-0.1	-0.5	0.2	0.0
62.7	0.0	0.0	-0.1	0.0
65.2	0.0	0.0	0.1	0.0
67.8	-0.1	-0.5	0.0	0.0
70.3	0.0	0.0	-0.1	0.0
72.8	-0.1	-0.5	0.0	0.0
75.3	-1.6	-12.3	-0.3	0.0

Prolog Compiler

Time (sec)	Net Rate (kbytes/sec)			
	cons	vector	number	other
2.7	55.7	2.4	0.0	0.5
5.4	147.1	113.0	0.0	-0.0
8.1	44.5	0.0	0.0	0.0
10.8	30.3	0.0	0.0	0.0
13.5	84.4	0.0	0.0	0.0
16.2	60.2	0.0	0.0	0.0
18.9	56.0	0.0	0.0	0.0
21.6	12.5	-0.1	-0.0	-0.1
24.3	42.1	-1.9	0.0	-0.0
27.0	-345.2	0.0	0.0	0.0
29.8	-57.3	-36.0	0.0	0.0
32.5	41.6	0.0	0.0	0.0
35.2	28.8	-0.0	0.0	0.0
37.9	61.0	-2.7	0.0	0.0
40.6	61.5	-8.7	0.0	0.0
43.3	42.1	0.0	0.0	0.0
46.0	11.0	-2.2	0.0	0.0
48.7	50.7	-4.6	0.0	-0.0
51.4	-318.4	0.0	0.0	0.0
54.1	26.5	-3.8	0.0	0.0
56.8	15.0	-1.3	0.0	0.0
59.5	66.5	-1.2	0.0	0.0
62.2	39.7	-1.3	0.0	0.0
64.9	65.1	-0.6	0.0	0.0
67.6	42.1	-0.7	0.0	0.0
70.3	29.0	-2.0	0.0	0.0
73.0	32.0	-0.8	0.0	0.0
75.7	26.0	-19.5	0.0	0.0
78.4	-6.4	-21.1	0.0	-0.1
81.1	-15.7	-0.0	0.0	-0.3

Table C.8: Net Allocation Rates as a Function of Time.

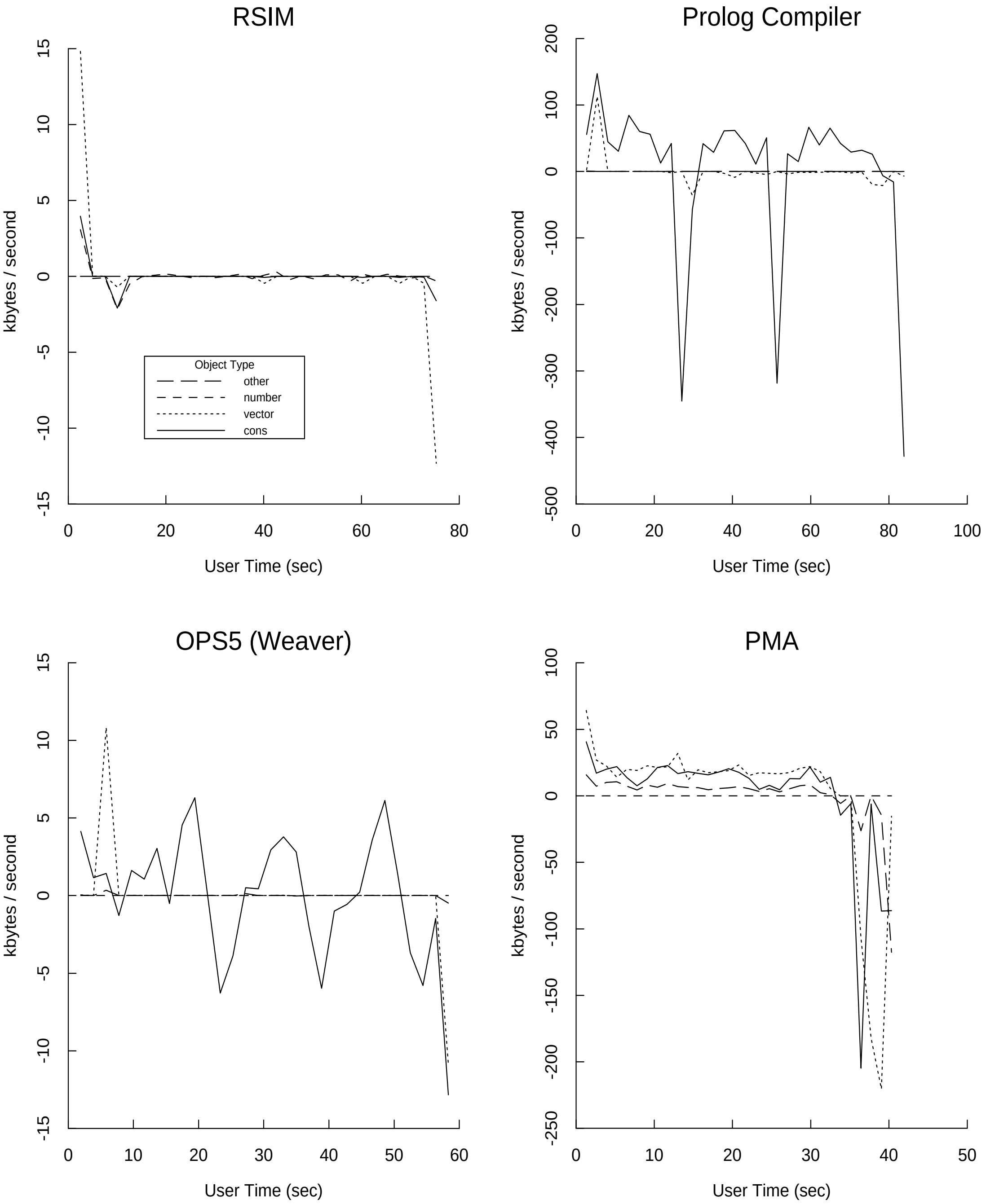

Figure C.6: Net Allocation Rates as a Function of Time.

OPS5 (Weaver)

Time	Net Rate (kbytes/sec)			
(sec)	cons	vector	number	other
1.9	4.1	0.0	0.0	0.0
3.9	1.2	0.0	0.0	0.0
5.8	1.4	10.8	0.0	0.3
7.8	-1.3	0.0	0.0	0.0
9.7	1.6	0.0	0.0	0.0
11.7	1.1	0.0	0.0	0.0
13.6	3.0	0.0	0.0	0.0
15.5	-0.5	0.0	0.0	0.0
17.5	4.5	0.0	0.0	0.0
19.4	6.3	0.0	0.0	0.0
21.4	0.0	0.0	0.0	0.0
23.3	-6.3	0.0	0.0	0.0
25.3	-3.9	0.0	0.0	0.0
27.2	0.5	0.0	0.0	0.1
29.2	0.4	0.0	0.0	0.0
31.1	3.0	0.0	0.0	0.0
33.0	3.8	0.0	0.0	0.0
35.0	2.8	-0.0	-0.0	-0.0
36.9	-2.0	0.0	0.0	0.0
38.9	-6.0	0.0	0.0	0.0
40.8	-1.0	0.0	0.0	0.0
42.8	-0.6	0.0	0.0	0.0
44.7	0.2	0.0	0.0	0.0
46.6	3.6	0.0	0.0	0.0
48.6	6.1	0.0	0.0	0.0
50.5	1.4	0.0	0.0	0.0
52.5	-3.7	0.0	0.0	0.0
54.4	-5.8	0.0	0.0	0.0
56.4	-1.5	0.0	0.0	0.0
58.3	-12.8	-10.9	0.0	-0.5

PMA

Time	Net Rate (kbytes/sec)			
(sec)	cons	vector	number	other
1.3	40.6	64.2	0.0	15.8
2.6	17.2	27.0	0.0	7.2
3.9	20.0	22.5	0.0	10.1
5.2	22.0	14.1	0.0	10.6
6.5	13.7	19.9	0.0	7.3
7.8	7.6	19.1	0.0	4.4
9.1	12.9	22.5	0.0	8.0
10.4	21.4	21.4	0.0	6.4
11.7	22.7	21.9	0.0	9.3
13.0	16.8	32.0	0.0	6.9
14.3	18.2	12.0	0.0	6.3
15.6	17.0	19.7	0.0	6.2
16.9	15.8	17.4	0.0	4.6
18.2	17.9	18.2	0.0	5.6
19.5	20.5	19.0	0.0	6.0
20.8	17.8	23.3	0.0	6.9
22.1	13.1	15.4	0.0	5.4
23.4	5.0	17.5	0.0	3.3
24.7	8.0	16.9	0.0	5.4
26.0	4.7	16.6	0.0	3.1
27.3	12.9	17.5	-0.0	5.4
28.6	12.8	20.8	0.0	7.7
29.9	21.7	22.0	0.0	8.5
31.2	10.2	18.1	0.0	2.3
32.5	14.0	5.5	-0.0	0.9
33.8	-14.5	-0.1	0.0	-5.7
35.1	-5.9	0.0	0.0	0.0
36.4	-204.6	-106.7	0.0	-26.2
37.7	-6.2	-182.1	0.0	0.0
39.0	-86.6	-220.3	-0.0	-14.3

Table C.9: Net Allocation Rates as a Function of Time.

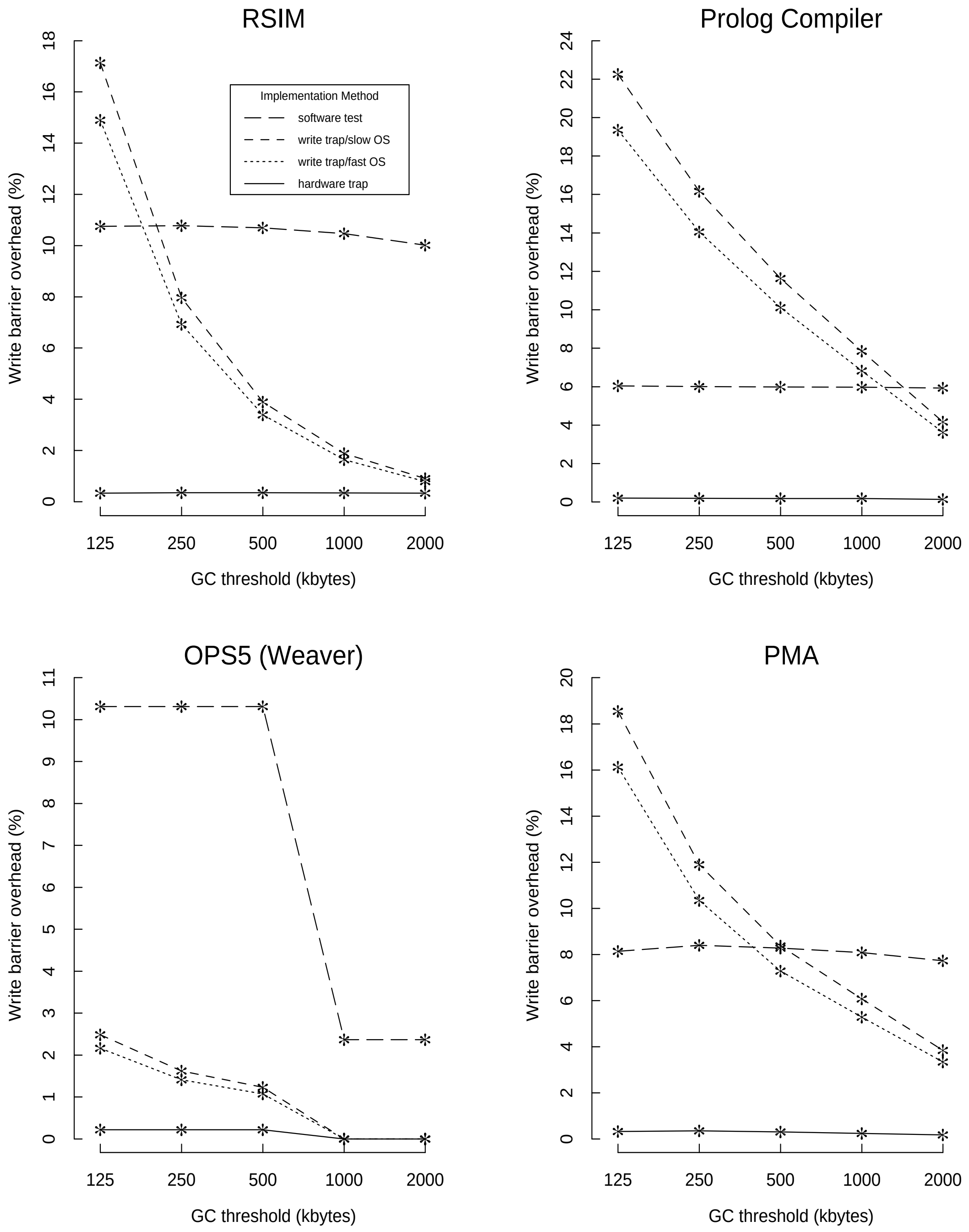

Figure C.7: CPU Overhead for Write Barrier Implementations. Stop-and-copy is the garbage collection algorithm used in all cases.

RSIM

Implementation Method	Write Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	0.3	0.3	0.4	0.3	0.3
write trap/fast OS	14.9	6.9	3.4	1.6	0.8
write trap/slow OS	17.1	8.0	3.9	1.9	0.9
software test	10.8	10.8	10.7	10.5	10.0

Prolog Compiler

Implementation Method	Write Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	0.2	0.2	0.2	0.2	0.1
write trap/fast OS	19.4	14.1	10.1	6.8	3.6
write trap/slow OS	22.3	16.2	11.6	7.9	4.2
software test	6.0	6.0	6.0	6.0	5.9

OPS5 (Weaver)

Implementation Method	Write Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	0.2	0.2	0.2	0.0	0.0
write trap/fast OS	2.2	1.4	1.1	0.0	0.0
write trap/slow OS	2.5	1.6	1.2	0.0	0.0
software test	10.3	10.3	10.3	2.4	2.4

PMA

Implementation Method	Write Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	0.3	0.4	0.3	0.2	0.2
write trap/fast OS	16.1	10.3	7.3	5.3	3.3
write trap/slow OS	18.5	11.9	8.4	6.1	3.8
software test	8.1	8.4	8.3	8.1	7.7

Table C.10: CPU Overhead for Write Barrier Implementations. Stop-and-copy is the garbage collection algorithm used in all cases.

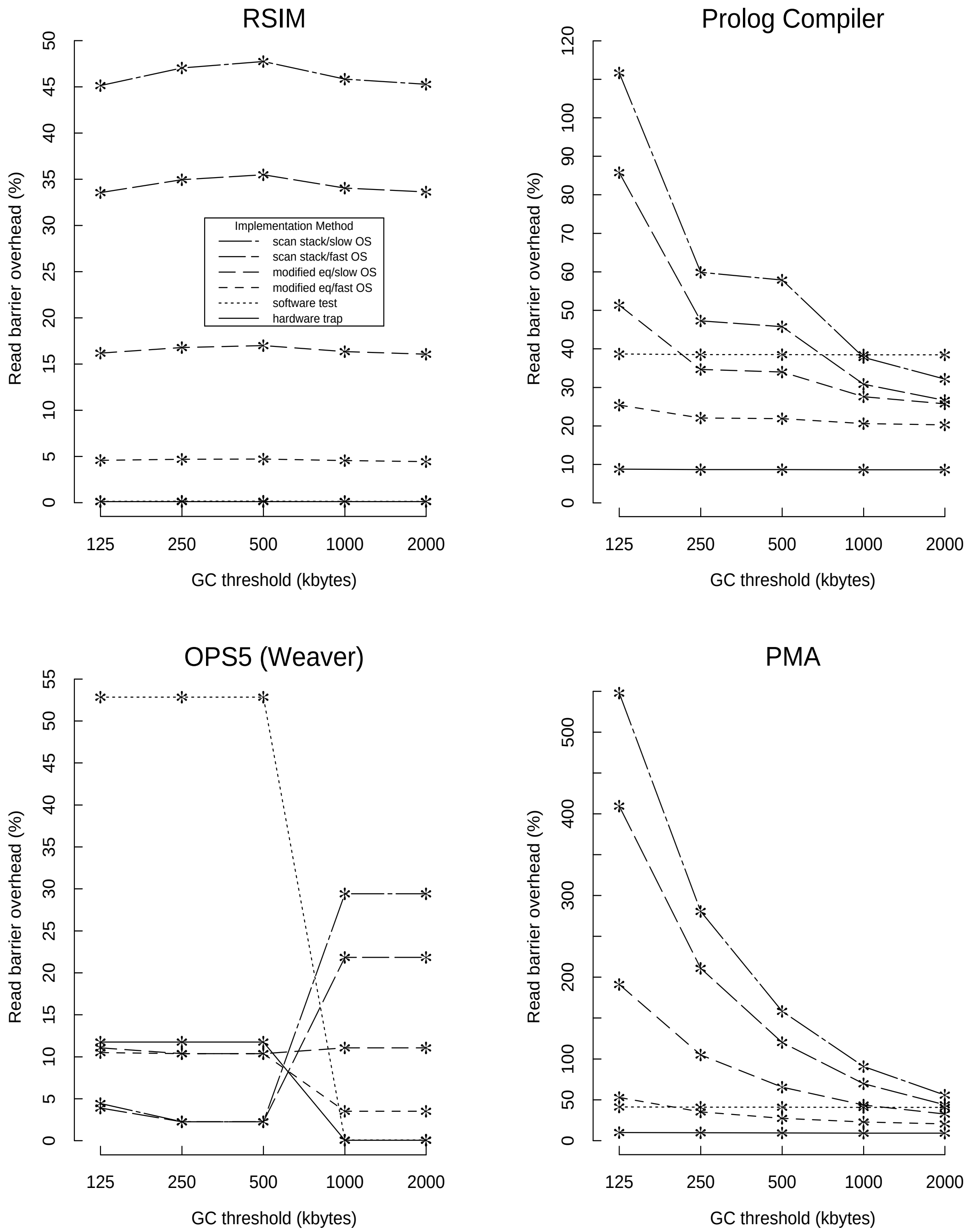

Figure C.8: CPU Overhead for Read Barrier Implementations. The results indicate the overhead of the read barrier for an incremental copying algorithm.

RSIM

Implementation Method	Read Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	0.1	0.1	0.1	0.1	0.1
software test	0.1	0.1	0.1	0.1	0.1
modified eq/fast OS	4.6	4.7	4.7	4.6	4.4
modified eq/slow OS	16.2	16.8	17.0	16.3	16.1
scan stack/fast OS	33.6	35.0	35.5	34.1	33.6
scan stack/slow OS	45.2	47.1	47.8	45.8	45.3

Prolog Compiler

Implementation Method	Read Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	8.8	8.6	8.6	8.6	8.6
software test	38.7	38.5	38.5	38.5	38.4
modified eq/fast OS	25.4	22.0	21.9	20.6	20.3
modified eq/slow OS	51.3	34.7	34.0	27.6	25.7
scan stack/fast OS	85.7	47.3	45.8	30.8	26.6
scan stack/slow OS	111.7	59.9	57.9	37.8	32.1

OPS5 (Weaver)

Implementation Method	Read Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	11.7	11.7	11.7	0.1	0.1
software test	52.8	52.8	52.8	0.1	0.1
modified eq/fast OS	10.5	10.4	10.4	3.5	3.5
modified eq/slow OS	11.1	10.4	10.4	11.1	11.1
scan stack/fast OS	3.9	2.3	2.3	21.8	21.8
scan stack/slow OS	4.4	2.3	2.3	29.4	29.4

PMA

Implementation Method	Read Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	10.0	9.5	9.3	9.2	9.1
software test	41.1	41.0	40.8	40.7	40.6
modified eq/fast OS	52.9	35.2	27.2	22.8	20.5
modified eq/slow OS	191.1	104.8	65.4	43.6	32.4
scan stack/fast OS	409.6	210.9	120.0	69.7	43.8
scan stack/slow OS	547.8	280.5	158.3	90.6	55.7

Table C.11: CPU Overhead for Read Barrier Implementations. The results indicate the overhead of the read barrier for an incremental copying algorithm.

RSIM (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	13.6	13.7	13.6	13.5	13.3
scan	14.9	14.4	14.1	13.8	13.5
forward	15.2	14.6	14.2	13.9	13.5
transport	16.0	15.1	14.5	14.1	13.7
update	16.3	15.2	14.6	14.1	13.7

Prolog Compiler (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	6.9	6.9	6.9	6.8	6.7
scan	23.5	22.3	18.7	18.2	16.4
forward	29.4	28.0	23.1	22.3	20.1
transport	41.6	39.3	31.7	30.7	27.3
update	45.6	43.0	34.6	33.4	29.7

OPS5 (Weaver) (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	0.2	0.1	0.1	—	—
scan	0.4	0.3	0.1	—	—
forward	0.5	0.3	0.2	—	—
transport	0.6	0.4	0.2	—	—
update	0.7	0.4	0.2	—	—

PMA (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	4.7	4.3	4.2	4.0	3.7
scan	35.3	22.1	16.4	13.4	11.1
forward	39.2	24.8	18.7	15.5	13.0
transport	48.6	31.6	24.3	20.6	17.5
update	51.1	33.4	25.7	21.8	18.6

Table C.12: Cumulative CPU Overhead of Copying Collection. The algorithm used is stop-and-copy collection. Results for incremental copying are similar.

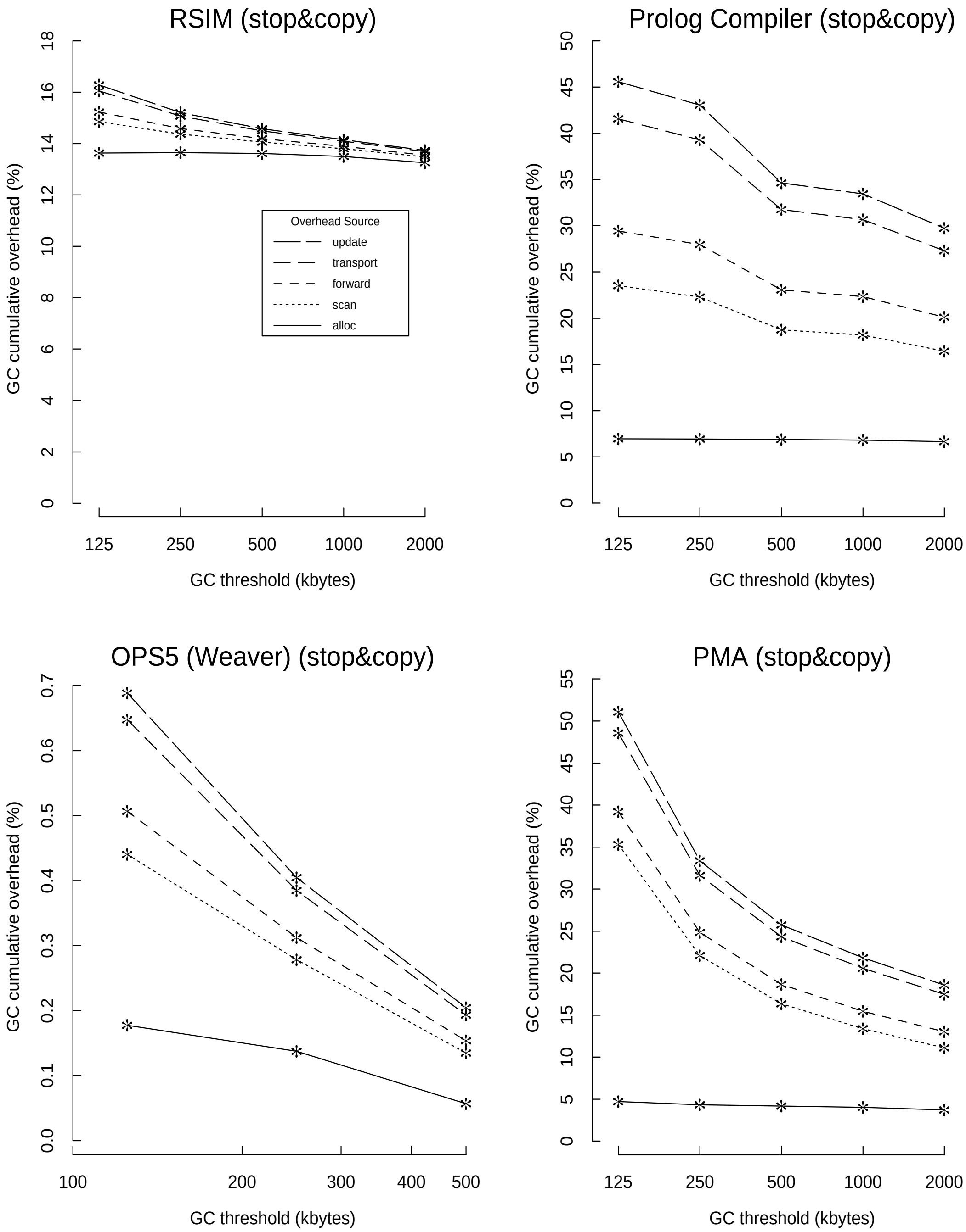

Figure C.9: Cumulative CPU Overhead of Copying Collection. The algorithm used is stop-and-copy collection. Results for incremental copying are similar.

RSIM (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	15.6	15.6	15.5	15.2	14.6
sweep	21.5	21.4	21.1	20.7	19.9
stack	22.0	21.7	21.3	20.8	20.0
type	22.9	22.1	21.6	21.0	20.1
marking	23.7	22.6	21.9	21.2	20.2
relocate	23.8	22.7	21.9	21.2	20.3

Prolog Compiler (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	9.2	9.1	9.0	8.6	8.3
sweep	17.3	16.8	16.2	15.5	14.7
stack	22.4	21.8	20.8	19.6	18.4
type	30.8	30.1	28.4	26.5	24.7
marking	38.3	37.4	35.3	32.6	30.6
relocate	39.1	38.2	36.0	33.2	31.1

OPS5 (Weaver) (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	0.2	0.1	0.0	0.0	0.0
sweep	0.3	0.1	0.0	0.0	0.0
stack	0.4	0.1	0.0	0.0	0.0
type	0.6	0.1	0.0	0.0	0.0
marking	0.7	0.1	0.0	0.0	0.0
relocate	0.8	0.1	0.0	0.0	0.0

PMA (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	4.8	4.4	4.3	3.9	3.4
sweep	9.2	8.1	7.8	7.1	5.8
stack	22.1	15.5	12.7	10.6	8.1
type	42.1	26.8	20.5	15.9	12.0
marking	59.5	36.1	26.9	19.9	15.3
relocate	61.5	37.7	28.3	21.1	16.4

Table C.13: Cumulative CPU Overhead of Mark-and-Sweep Collection.

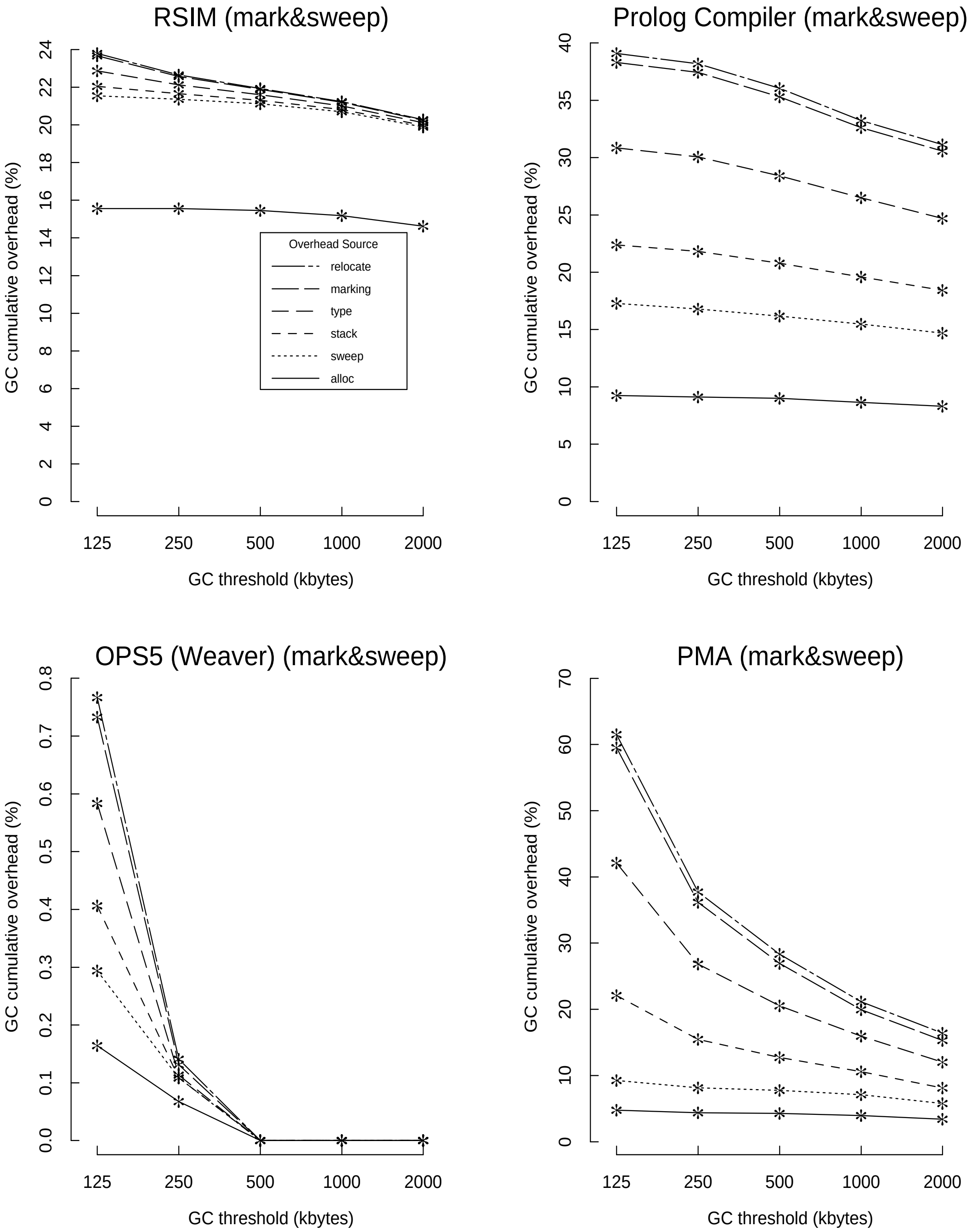

Figure C.10: Cumulative CPU Overhead of Mark-and-Sweep Collection.

Prolog Compiler (incremental)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	39.0	30.4	28.0	26.1	22.4
write barrier	59.4	50.7	42.0	35.9	28.1
read barrier	110.7	85.4	76.1	63.4	53.8

PMA (incremental)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	46.8	29.3	21.7	17.4	13.9
write barrier	66.8	42.6	31.3	24.3	18.1
read barrier	257.9	147.4	96.7	67.9	50.5

Prolog Compiler (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	38.6	36.1	27.7	26.6	23.1
write barrier	60.9	52.3	39.4	34.5	27.2

PMA (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	46.4	29.0	21.6	17.8	14.9
write barrier	64.9	40.9	29.9	23.9	18.7

Prolog Compiler (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	39.1	38.2	36.0	33.2	31.1
write barrier	54.5	49.5	44.9	39.5	35.9
indirect vectors	54.8	49.9	45.3	39.9	36.2

PMA (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	61.5	37.7	28.3	21.1	16.4
write barrier	77.4	48.0	35.1	26.1	19.8
indirect vectors	80.0	50.6	37.7	28.7	22.4

Table C.14: Cumulative CPU Overhead for Three Algorithms.

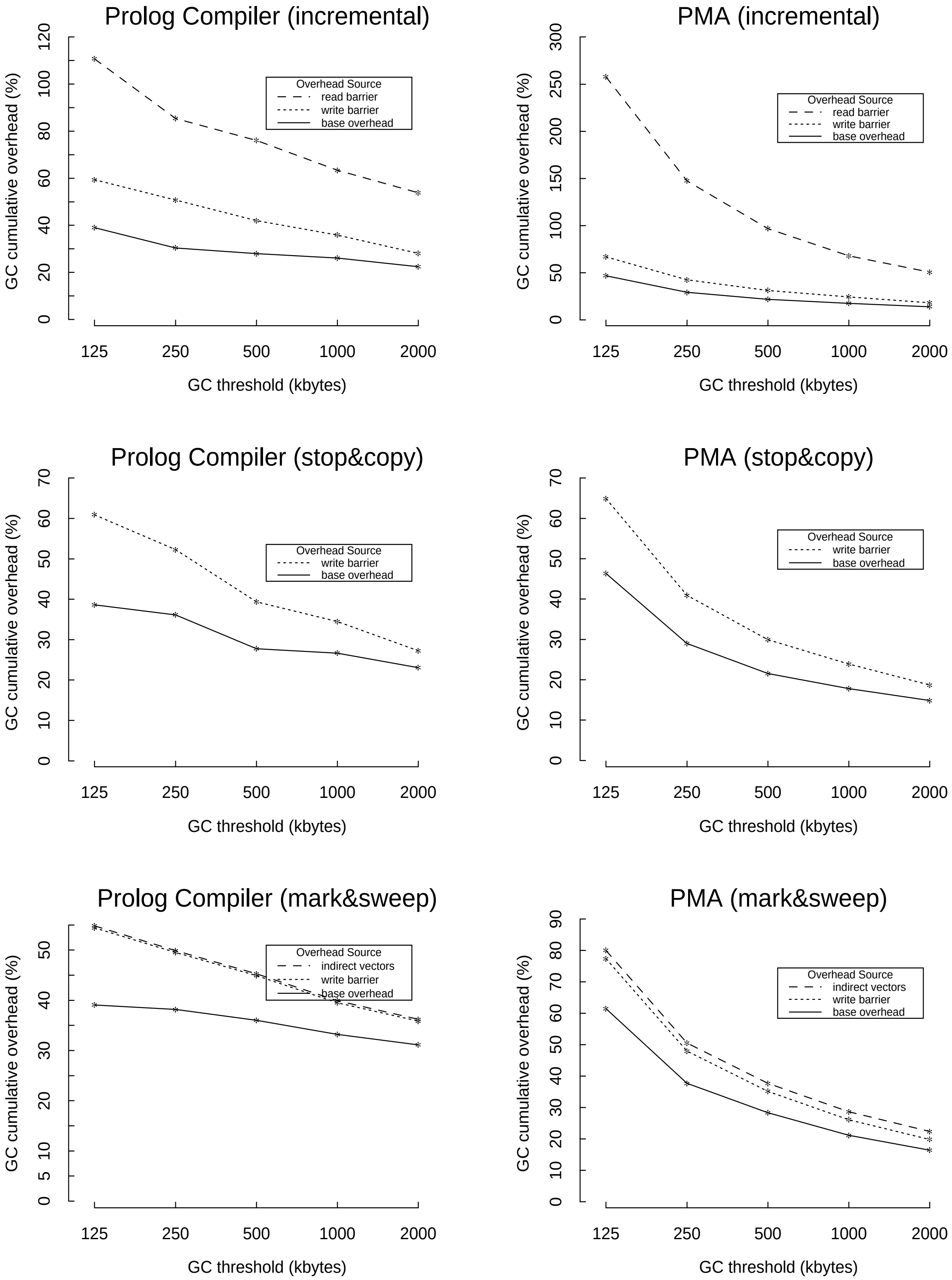

Figure C.11: Cumulative CPU Overhead for Three Algorithms.

Age	RSIM					Prolog Compiler				
	Fraction of references (%)					Fraction of references (%)				
(sec)	cons	vector	number	other	total	cons	vector	number	other	total
0.000004	43.9	0.3	95.6	20.6	38.2	21.1	46.0	55.6	16.5	22.1
0.000008	21.2	0.1	2.2	10.3	6.7	8.9	1.8	22.2	8.2	8.5
0.000016	15.1	0.1	2.2	19.9	4.9	1.9	2.7	22.2	16.2	2.4
0.000032	6.0	0.0	0.0	39.1	1.7	2.7	3.1	0.0	31.9	3.7
0.000064	0.1	0.0	0.0	9.8	0.0	2.8	0.1	0.0	24.0	3.3
0.000128	0.0	0.0	0.0	0.0	0.0	3.7	0.2	0.0	0.6	3.4
0.000256	0.1	0.0	0.0	0.2	0.0	3.7	0.8	0.0	0.9	3.5
0.000512	0.1	0.0	0.0	0.0	0.0	6.3	0.3	0.0	1.3	5.8
0.001024	0.0	0.0	0.0	0.0	0.0	7.4	1.0	0.0	0.2	6.9
0.002048	0.0	0.0	0.0	0.0	0.0	1.1	0.7	0.0	0.2	1.1
0.004096	0.0	0.1	0.0	0.0	0.0	2.8	0.3	0.0	0.0	2.6
0.008192	0.0	0.0	0.0	0.0	0.0	7.1	0.3	0.0	0.0	6.6
0.016384	0.1	0.0	0.0	0.0	0.0	8.6	0.2	0.0	0.0	7.9
0.032768	0.1	0.0	0.0	0.0	0.0	3.2	0.4	0.0	0.0	2.9
0.065536	0.1	0.0	0.0	0.0	0.0	0.9	0.5	0.0	0.0	0.9
0.131072	0.2	0.1	0.0	0.0	0.1	0.4	0.6	0.0	0.0	0.4
0.262144	0.0	0.0	0.0	0.0	0.0	0.4	0.8	0.0	0.0	0.4
0.524288	0.2	0.1	0.0	0.0	0.1	0.9	1.7	0.0	0.0	0.9
1.048580	0.3	0.1	0.0	0.0	0.1	1.3	2.3	0.0	0.0	1.3
2.097150	0.5	0.5	0.0	0.0	0.4	2.6	3.2	0.0	0.0	2.6
4.194300	0.3	1.4	0.0	0.0	0.7	3.8	3.4	0.0	0.0	3.6
8.388610	0.3	2.8	0.0	0.0	1.3	2.4	6.5	0.0	0.0	2.5
16.777200	0.7	5.6	0.0	0.0	2.7	1.5	7.0	0.0	0.0	1.7
33.554400	1.3	11.2	0.0	0.0	5.4	1.6	1.2	0.0	0.0	1.5
67.108900	2.7	22.4	0.0	0.0	10.8	1.3	5.5	0.0	0.0	1.5
134.218000	5.4	44.5	0.0	0.0	21.4	1.0	6.2	0.0	0.0	1.2
268.435000	1.3	10.6	0.0	0.0	5.1	0.5	3.1	0.0	0.0	0.6

Table C.15: Age Distribution of Objects Referenced by Object Type.

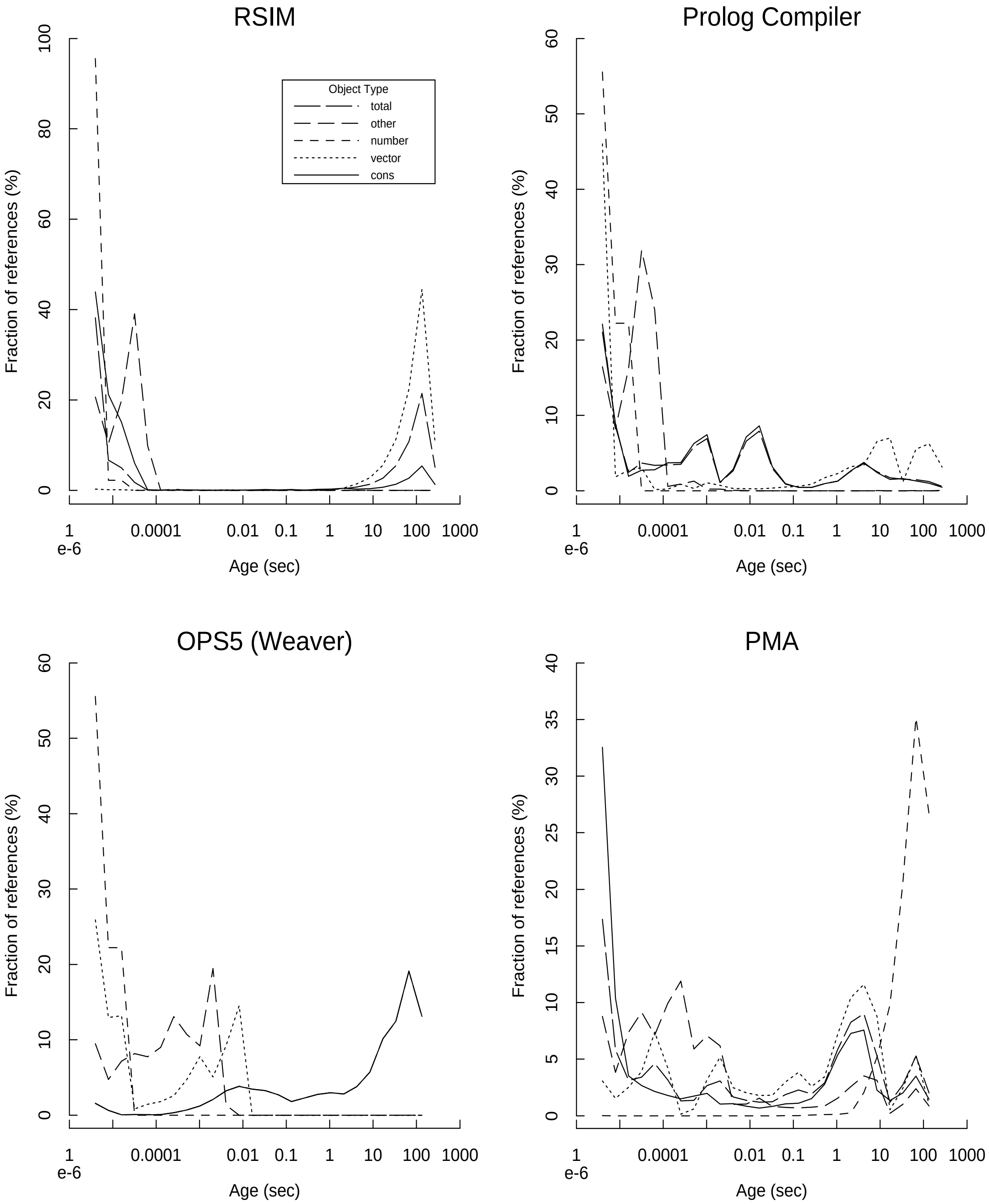

Figure C.12: Age Distribution of Objects Referenced by Object Type.

OPS5 (Weaver)

Age (sec)	Fraction of references (%)				
	cons	vector	number	other	total
0.000004	1.5	25.9	55.6	9.5	1.6
0.000008	0.6	13.0	22.2	4.7	0.6
0.000016	0.0	13.2	22.2	7.2	0.1
0.000032	0.1	0.8	0.0	8.2	0.1
0.000064	0.1	1.4	0.0	7.7	0.1
0.000128	0.1	1.8	0.0	9.0	0.1
0.000256	0.3	2.6	0.0	13.1	0.3
0.000512	0.7	4.7	0.0	10.7	0.7
0.001024	1.2	7.8	0.0	9.2	1.2
0.002048	2.1	5.1	0.0	19.5	2.1
0.004096	3.2	9.2	0.0	1.3	3.2
0.008192	3.8	14.5	0.0	0.0	3.8
0.016384	3.5	0.0	0.0	0.0	3.5
0.032768	3.2	0.0	0.0	0.0	3.2
0.065536	2.7	0.0	0.0	0.0	2.7
0.131072	1.8	0.0	0.0	0.0	1.8
0.262144	2.3	0.0	0.0	0.0	2.3
0.524288	2.7	0.0	0.0	0.0	2.7
1.048580	3.0	0.0	0.0	0.0	3.0
2.097150	2.8	0.0	0.0	0.0	2.8
4.194300	3.8	0.0	0.0	0.0	3.8
8.388610	5.7	0.0	0.0	0.0	5.7
16.777200	10.1	0.0	0.0	0.0	10.1
33.554400	12.5	0.0	0.0	0.0	12.4
67.108900	19.1	0.0	0.0	0.0	19.1
134.218000	13.1	0.0	0.0	0.0	13.1

PMA

Age (sec)	Fraction of references (%)				
	cons	vector	number	other	total
0.000004	32.5	3.1	0.0	8.8	17.3
0.000008	10.4	1.5	0.0	3.8	5.8
0.000016	3.5	2.5	0.0	7.4	3.1
0.000032	2.7	3.9	0.0	9.2	3.4
0.000064	2.1	7.4	0.0	7.1	4.6
0.000128	1.8	4.2	0.0	9.9	3.2
0.000256	1.5	0.1	0.0	11.9	1.3
0.000512	1.7	0.6	0.0	5.9	1.4
0.001024	2.0	3.2	0.0	7.1	2.7
0.002048	1.0	5.2	0.0	6.2	3.1
0.004096	1.1	2.5	0.0	1.0	1.7
0.008192	0.9	2.0	0.0	1.0	1.4
0.016384	0.7	1.8	0.0	1.6	1.2
0.032768	0.8	1.8	0.0	0.8	1.2
0.065536	1.1	3.0	0.0	0.7	1.9
0.131072	1.1	3.8	0.0	0.7	2.3
0.262144	1.5	2.6	0.1	0.7	1.9
0.524288	2.8	3.4	0.1	0.9	2.9
1.048580	5.4	7.2	0.1	1.5	5.9
2.097150	7.3	10.4	0.2	2.6	8.3
4.194300	7.6	11.6	2.1	3.5	9.0
8.388610	2.3	8.8	5.2	3.1	5.3
16.777200	1.3	0.5	9.8	0.2	1.2
33.554400	2.0	2.4	20.6	1.0	2.7
67.108900	3.5	5.3	35.2	2.4	5.3
134.218000	1.5	1.0	26.7	0.9	2.0

Table C.16: Age Distribution of Objects Referenced by Object Type.

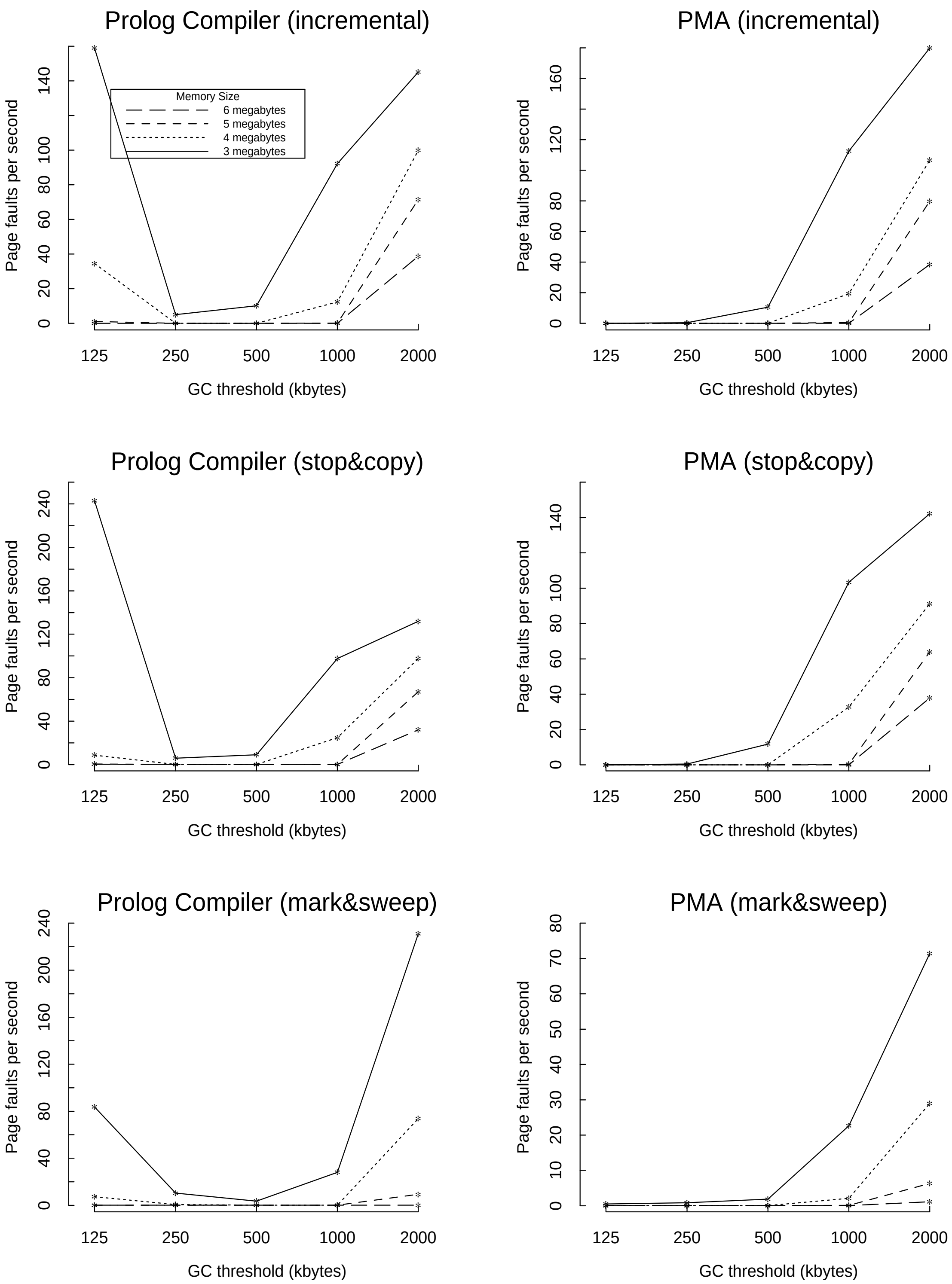

Figure C.13: Page Fault Rates for Different Collection Algorithms.

Prolog Compiler (incremental)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	159.0	5.0	10.2	92.3	145.2
4 megabytes	34.5	0.0	0.0	12.3	99.9
5 megabytes	1.0	0.0	0.0	0.2	71.4
6 megabytes	0.0	0.0	0.0	0.0	38.7

PMA (incremental)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	0.0	0.4	10.6	112.5	179.9
4 megabytes	0.0	0.0	0.1	19.4	106.5
5 megabytes	0.0	0.0	0.0	0.5	79.7
6 megabytes	0.0	0.0	0.0	0.1	38.4

Prolog Compiler (stop&copy)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	243.0	5.9	8.9	97.6	131.9
4 megabytes	8.7	0.0	0.0	24.8	97.7
5 megabytes	0.4	0.0	0.0	0.1	67.1
6 megabytes	0.3	0.0	0.0	0.1	32.1

PMA (stop&copy)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	0.0	0.5	11.7	103.4	142.1
4 megabytes	0.0	0.1	0.1	32.7	91.3
5 megabytes	0.0	0.1	0.0	0.4	63.9
6 megabytes	0.0	0.1	0.0	0.1	37.9

Prolog Compiler (mark&sweep)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	83.8	10.4	3.5	28.1	231.0
4 megabytes	7.3	0.6	0.0	0.1	73.8
5 megabytes	0.0	0.0	0.0	0.0	9.4
6 megabytes	0.0	0.0	0.0	0.0	0.1

PMA (mark&sweep)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	0.5	0.9	1.9	22.7	71.4
4 megabytes	0.0	0.0	0.0	2.1	29.0
5 megabytes	0.0	0.0	0.0	0.1	6.3
6 megabytes	0.0	0.0	0.0	0.0	1.1

Table C.17: Page Fault Rates for Different Collection Algorithms.

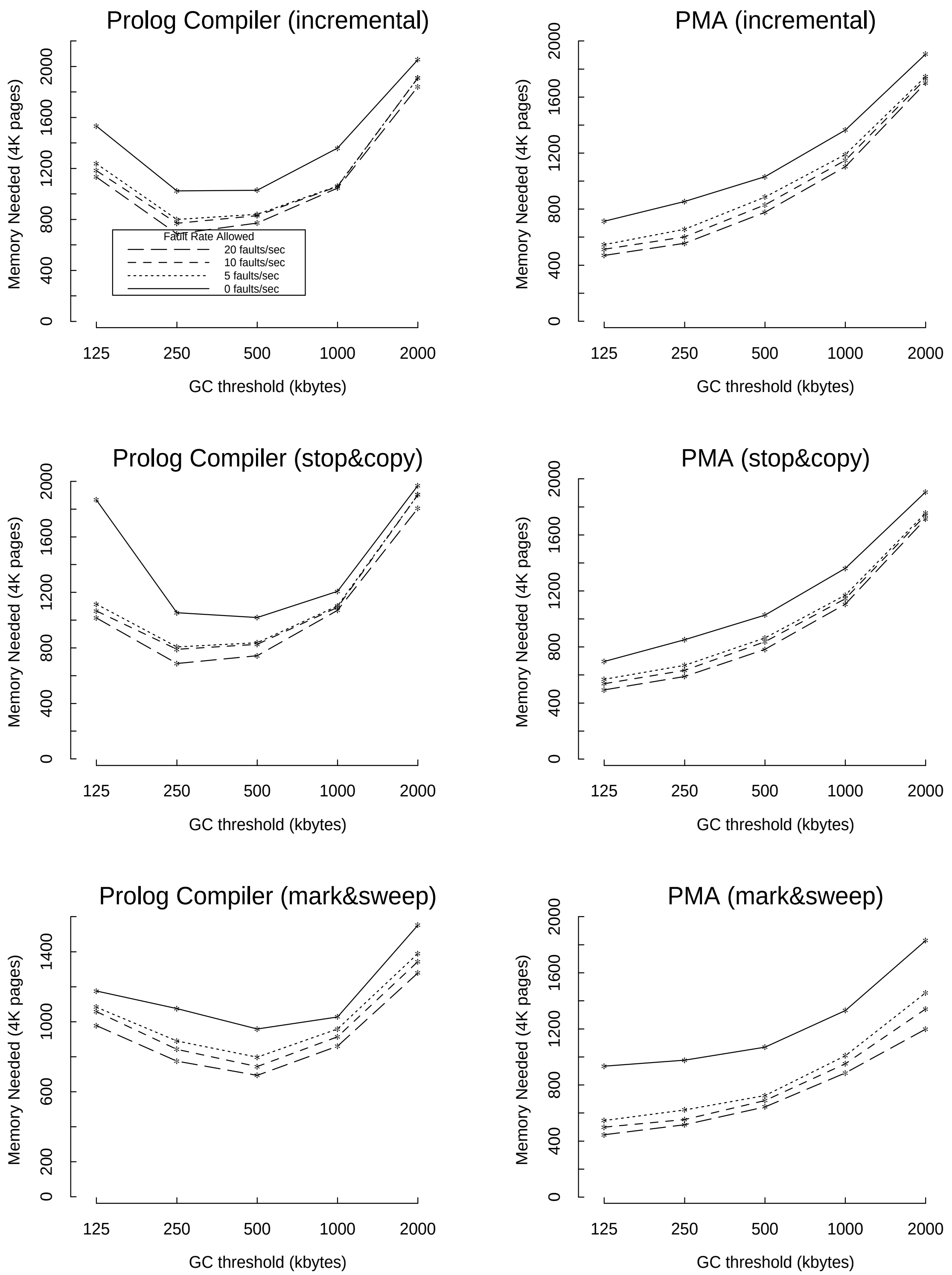

Figure C.14: Memory Sizes Required for Different Collection Algorithms.

Prolog Compiler (incremental)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	1533	1024	1028	1359	2053
5 faults/sec	1237	800	840	1062	1911
10 faults/sec	1183	769	830	1058	1911
20 faults/sec	1132	686	772	1046	1841

PMA (incremental)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	713	855	1031	1364	1907
5 faults/sec	546	657	886	1190	1746
10 faults/sec	513	602	831	1148	1729
20 faults/sec	470	556	778	1103	1701

Prolog Compiler (stop&copy)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	1866	1053	1019	1208	1970
5 faults/sec	1116	807	837	1103	1904
10 faults/sec	1066	790	826	1093	1904
20 faults/sec	1016	687	744	1071	1806

PMA (stop&copy)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	696	852	1028	1361	1906
5 faults/sec	570	669	865	1170	1757
10 faults/sec	538	633	837	1144	1741
20 faults/sec	493	589	784	1106	1714

Prolog Compiler (mark&sweep)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	1176	1075	959	1027	1551
5 faults/sec	1083	890	798	959	1389
10 faults/sec	1058	842	744	914	1344
20 faults/sec	978	775	694	861	1279

PMA (mark&sweep)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	934	977	1070	1330	1830
5 faults/sec	546	622	725	1009	1458
10 faults/sec	499	553	689	952	1342
20 faults/sec	445	517	644	886	1198

Table C.18: Memory Sizes Required for Different Collection Algorithms.

RSIM (stop&copy)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	7.30	7.34	7.14	7.13	7.14
128 kilobytes	6.09	6.03	6.00	5.99	5.97
256 kilobytes	3.05	5.38	5.40	5.39	5.39
512 kilobytes	3.03	3.85	5.02	5.08	5.08
1 megabyte	3.02	3.85	4.30	4.84	4.91
2 megabytes	1.11	0.58	0.43	0.76	4.75

Prolog Compiler (stop&copy)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	4.72	4.77	4.90	5.07	5.37
128 kilobytes	4.32	4.41	4.60	4.79	5.07
256 kilobytes	2.86	4.05	4.28	4.46	4.77
512 kilobytes	2.54	3.02	3.90	4.15	4.40
1 megabyte	2.12	2.76	3.15	3.80	4.03
2 megabytes	0.72	0.83	1.05	2.00	3.56

OPS5 (Weaver) (stop&copy)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	2.96	2.89	2.83	2.99	2.93
128 kilobytes	1.59	1.70	1.85	1.94	1.91
256 kilobytes	0.95	0.95	1.11	1.20	1.18
512 kilobytes	0.58	0.57	0.73	0.78	0.77
1 megabyte	0.32	0.29	0.33	0.33	0.33
2 megabytes	0.20	0.17	0.24	0.24	0.24

PMA (stop&copy)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	7.31	6.43	6.18	6.12	6.45
128 kilobytes	6.29	5.67	5.48	5.47	5.71
256 kilobytes	4.83	4.85	4.77	4.88	5.09
512 kilobytes	3.94	3.80	4.09	4.23	4.43
1 megabyte	3.65	3.47	3.32	3.69	3.92
2 megabytes	1.89	1.42	1.22	2.01	3.35

Table C.19: Cache Miss Rates for Stop-and-Copy Collection.

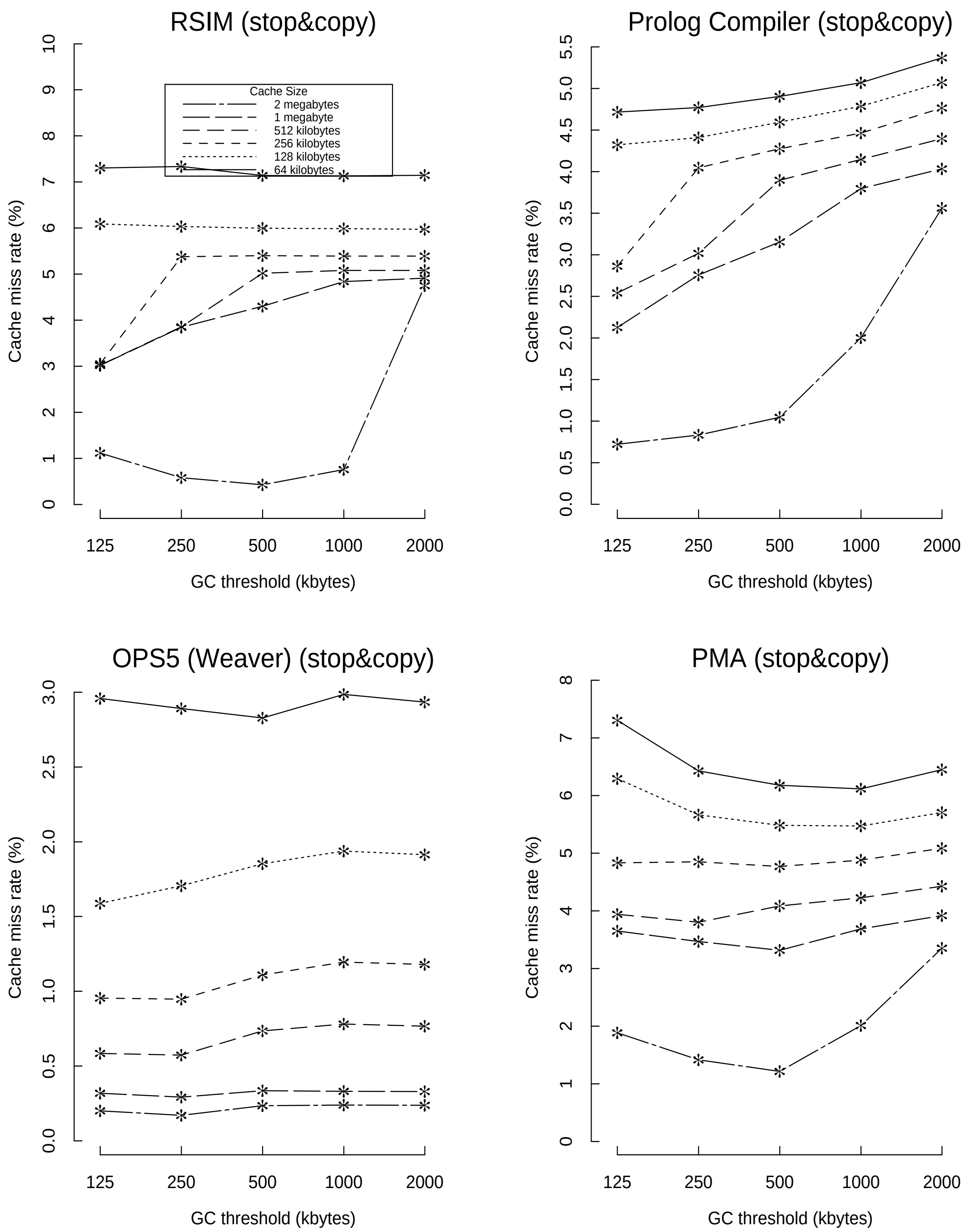

Figure C.15: Cache Miss Rates for Stop-and-Copy Collection.

RSIM (mark&sweep)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	5.35	5.35	4.94	5.38	4.25
128 kilobytes	1.60	2.54	3.13	3.20	3.12
256 kilobytes	1.40	0.88	1.89	2.76	2.71
512 kilobytes	1.27	0.70	0.68	1.63	2.52
1 megabyte	0.05	0.05	0.08	0.31	1.44
2 megabytes	0.05	0.05	0.08	0.13	0.24

Prolog Compiler (mark&sweep)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	5.16	5.03	5.15	5.21	5.09
128 kilobytes	3.73	4.44	4.72	4.80	4.71
256 kilobytes	2.50	2.87	4.23	4.45	4.40
512 kilobytes	1.82	1.57	2.68	3.98	4.03
1 megabyte	1.52	1.09	1.42	2.39	3.62
2 megabytes	1.25	0.90	1.04	1.49	2.08

OPS5 (Weaver) (mark&sweep)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	3.07	3.23	3.58	3.54	3.27
128 kilobytes	1.91	1.92	2.31	2.29	1.96
256 kilobytes	1.13	1.10	1.43	1.44	1.05
512 kilobytes	0.59	0.60	1.01	0.97	0.63
1 megabyte	0.28	0.30	0.71	0.68	0.36
2 megabytes	0.12	0.14	0.55	0.52	0.21

PMA (mark&sweep)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	6.69	5.85	5.68	5.58	5.58
128 kilobytes	5.84	5.06	4.91	4.93	4.97
256 kilobytes	4.26	4.12	4.22	4.29	4.37
512 kilobytes	2.43	2.66	3.23	3.59	3.82
1 megabyte	1.08	1.47	1.97	2.57	3.15
2 megabytes	0.87	1.18	1.46	1.92	2.31

Table C.20: Cache Miss Rates for Mark-and-Sweep Collection.

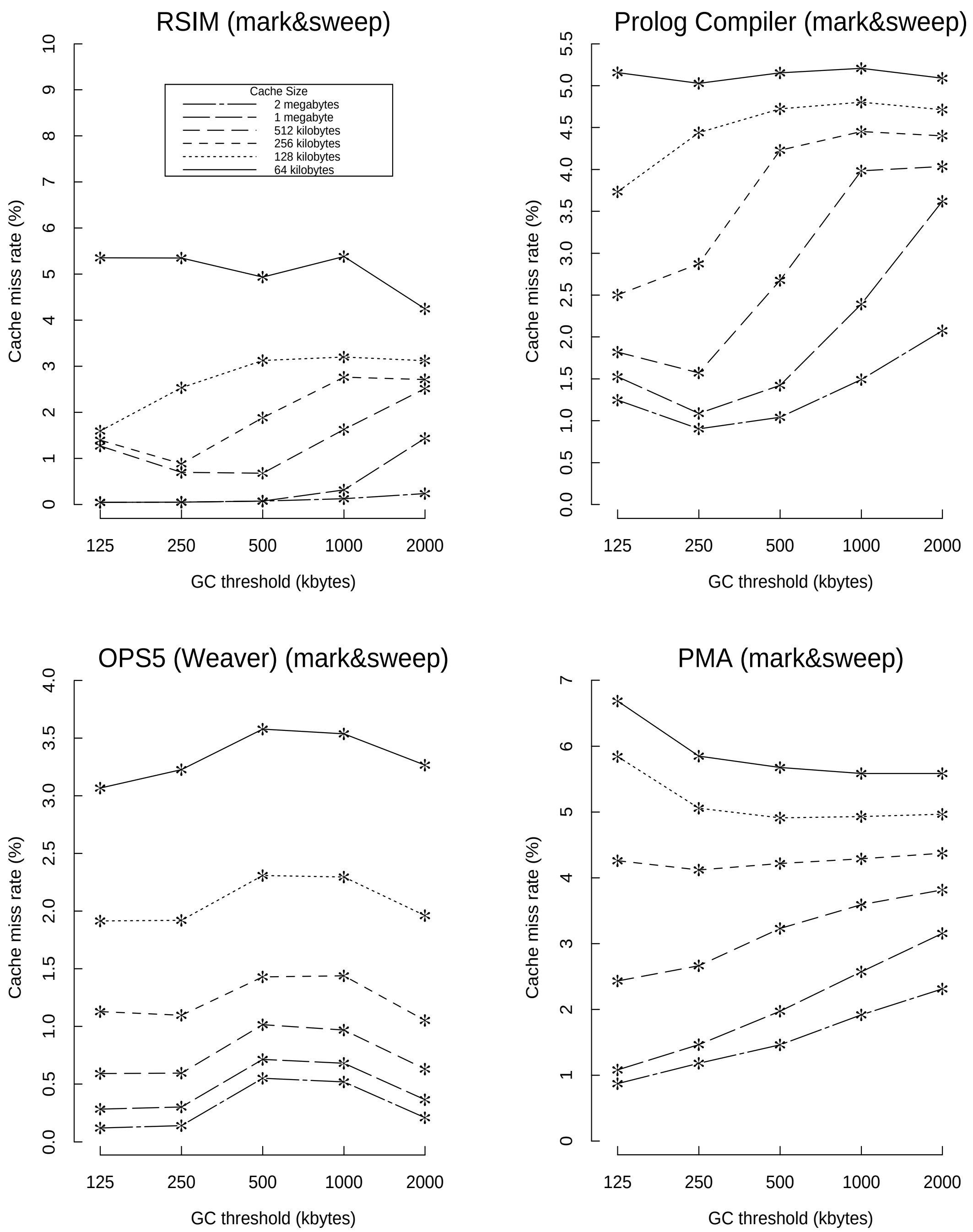

Figure C.16: Cache Miss Rates for Mark-and-Sweep Collection.

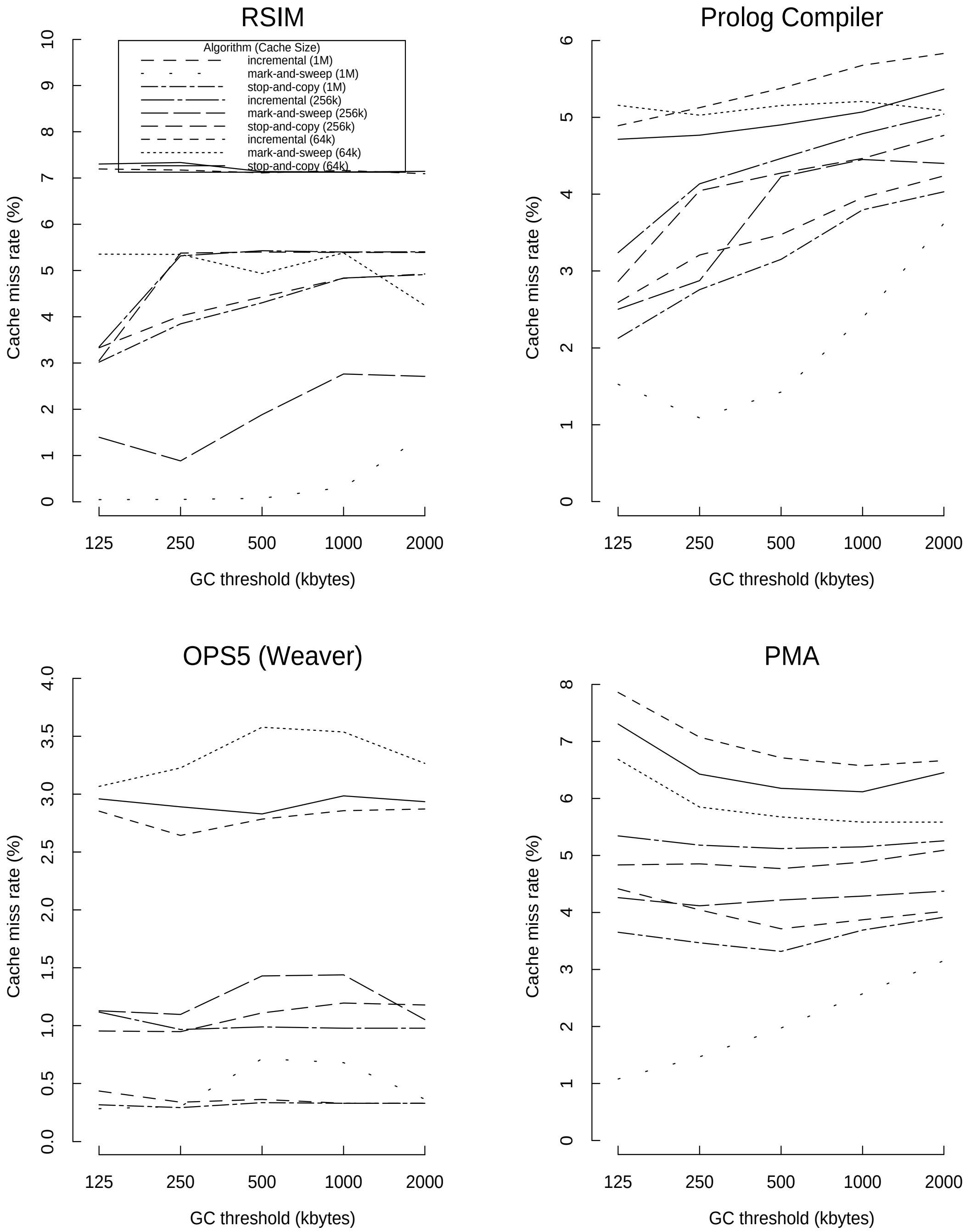

Figure C.17: Cache Miss Rates for Three Collection Algorithms.

RSIM

Algorithm (Cache Size)	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (64k)	7.30	7.34	7.14	7.13	7.14
mark-and-sweep (64k)	5.35	5.35	4.94	5.38	4.25
incremental (64k)	7.20	7.17	7.11	7.16	7.09
stop-and-copy (256k)	3.05	5.38	5.40	5.39	5.39
mark-and-sweep (256k)	1.40	0.88	1.89	2.76	2.71
incremental (256k)	3.35	5.32	5.43	5.40	5.41
stop-and-copy (1M)	3.02	3.85	4.30	4.84	4.91
mark-and-sweep (1M)	0.05	0.05	0.08	0.31	1.44
incremental (1M)	3.33	4.02	4.43	4.84	4.93

Prolog Compiler

Algorithm (Cache Size)	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (64k)	4.72	4.77	4.90	5.07	5.37
mark-and-sweep (64k)	5.16	5.03	5.15	5.21	5.09
incremental (64k)	4.89	5.13	5.38	5.68	5.83
stop-and-copy (256k)	2.86	4.05	4.28	4.46	4.77
mark-and-sweep (256k)	2.50	2.87	4.23	4.45	4.40
incremental (256k)	3.24	4.14	4.46	4.79	5.04
stop-and-copy (1M)	2.12	2.76	3.15	3.80	4.03
mark-and-sweep (1M)	1.52	1.09	1.42	2.39	3.62
incremental (1M)	2.59	3.21	3.47	3.96	4.24

OPS5 (Weaver)

Algorithm (Cache Size)	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (64k)	2.96	2.89	2.83	2.99	2.93
mark-and-sweep (64k)	3.07	3.23	3.58	3.54	3.27
incremental (64k)	2.85	2.64	2.78	2.86	2.87
stop-and-copy (256k)	0.95	0.95	1.11	1.20	1.18
mark-and-sweep (256k)	1.13	1.10	1.43	1.44	1.05
incremental (256k)	1.12	0.97	0.99	0.98	0.98
stop-and-copy (1M)	0.32	0.29	0.33	0.33	0.33
mark-and-sweep (1M)	0.28	0.30	0.71	0.68	0.36
incremental (1M)	0.44	0.34	0.36	0.33	0.33

PMA

Algorithm (Cache Size)	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (64k)	7.31	6.43	6.18	6.12	6.45
mark-and-sweep (64k)	6.69	5.85	5.68	5.58	5.58
incremental (64k)	7.86	7.08	6.71	6.57	6.67
stop-and-copy (256k)	4.83	4.85	4.77	4.88	5.09
mark-and-sweep (256k)	4.26	4.12	4.22	4.29	4.37
incremental (256k)	5.34	5.18	5.12	5.15	5.26
stop-and-copy (1M)	3.65	3.47	3.32	3.69	3.92
mark-and-sweep (1M)	1.08	1.47	1.97	2.57	3.15
incremental (1M)	4.42	4.05	3.71	3.87	4.02

Table C.21: Cache Miss Rates for Three Collection Algorithms.

Age (sec)	RSIM: Fraction surviving					Prolog Compiler: Fraction surviving				
	cons	vector	number	other	total	cons	vector	number	other	total
0.000004	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.000008	0.97	1.00	0.05	1.00	0.35	0.86	0.08	0.80	1.00	0.78
0.000016	0.96	0.06	0.00	1.00	0.32	0.83	0.04	0.20	1.00	0.76
0.000032	0.02	0.05	0.00	0.98	0.01	0.60	0.04	0.20	0.98	0.55
0.000064	0.01	0.05	0.00	0.02	0.01	0.57	0.04	0.20	0.02	0.51
0.000128	0.01	0.05	0.00	0.02	0.01	0.27	0.04	0.20	0.01	0.24
0.000256	0.01	0.05	0.00	0.00	0.01	0.27	0.04	0.20	0.01	0.24
0.000512	0.01	0.05	0.00	0.00	0.00	0.26	0.04	0.20	0.01	0.24
0.001024	0.01	0.05	0.00	0.00	0.00	0.18	0.04	0.20	0.00	0.16
0.002048	0.01	0.05	0.00	0.00	0.00	0.18	0.03	0.20	0.00	0.16
0.004096	0.01	0.05	0.00	0.00	0.00	0.18	0.03	0.20	0.00	0.16
0.008192	0.01	0.05	0.00	0.00	0.00	0.17	0.03	0.20	0.00	0.16
0.016384	0.01	0.05	0.00	0.00	0.00	0.16	0.03	0.20	0.00	0.14
0.032768	0.01	0.05	0.00	0.00	0.00	0.15	0.03	0.20	0.00	0.13
0.065536	0.01	0.05	0.00	0.00	0.00	0.15	0.03	0.20	0.00	0.13
0.131072	0.01	0.05	0.00	0.00	0.00	0.15	0.03	0.20	0.00	0.13
0.262144	0.01	0.05	0.00	0.00	0.00	0.15	0.03	0.20	0.00	0.13
0.524288	0.00	0.05	0.00	0.00	0.00	0.15	0.03	0.20	0.00	0.13
1.048580	0.00	0.05	0.00	0.00	0.00	0.14	0.03	0.20	0.00	0.13
2.097150	0.00	0.05	0.00	0.00	0.00	0.14	0.03	0.20	0.00	0.12
4.194300	0.00	0.05	0.00	0.00	0.00	0.14	0.03	0.20	0.00	0.12
8.388610	0.00	0.05	0.00	0.00	0.00	0.13	0.03	0.20	0.00	0.11
16.777200	0.00	0.04	0.00	0.00	0.00	0.11	0.03	0.20	0.00	0.10
33.554400	0.00	0.04	0.00	0.00	0.00	0.06	0.03	0.20	0.00	0.06
67.108900	0.00	0.04	0.00	0.00	0.00	0.01	0.02	0.00	0.00	0.01
134.218000	0.00	0.04	0.00	0.00	0.00	0.01	0.01	0.00	0.00	0.01
268.435000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table C.22: Survival Distribution of Objects Referenced by Object Type.

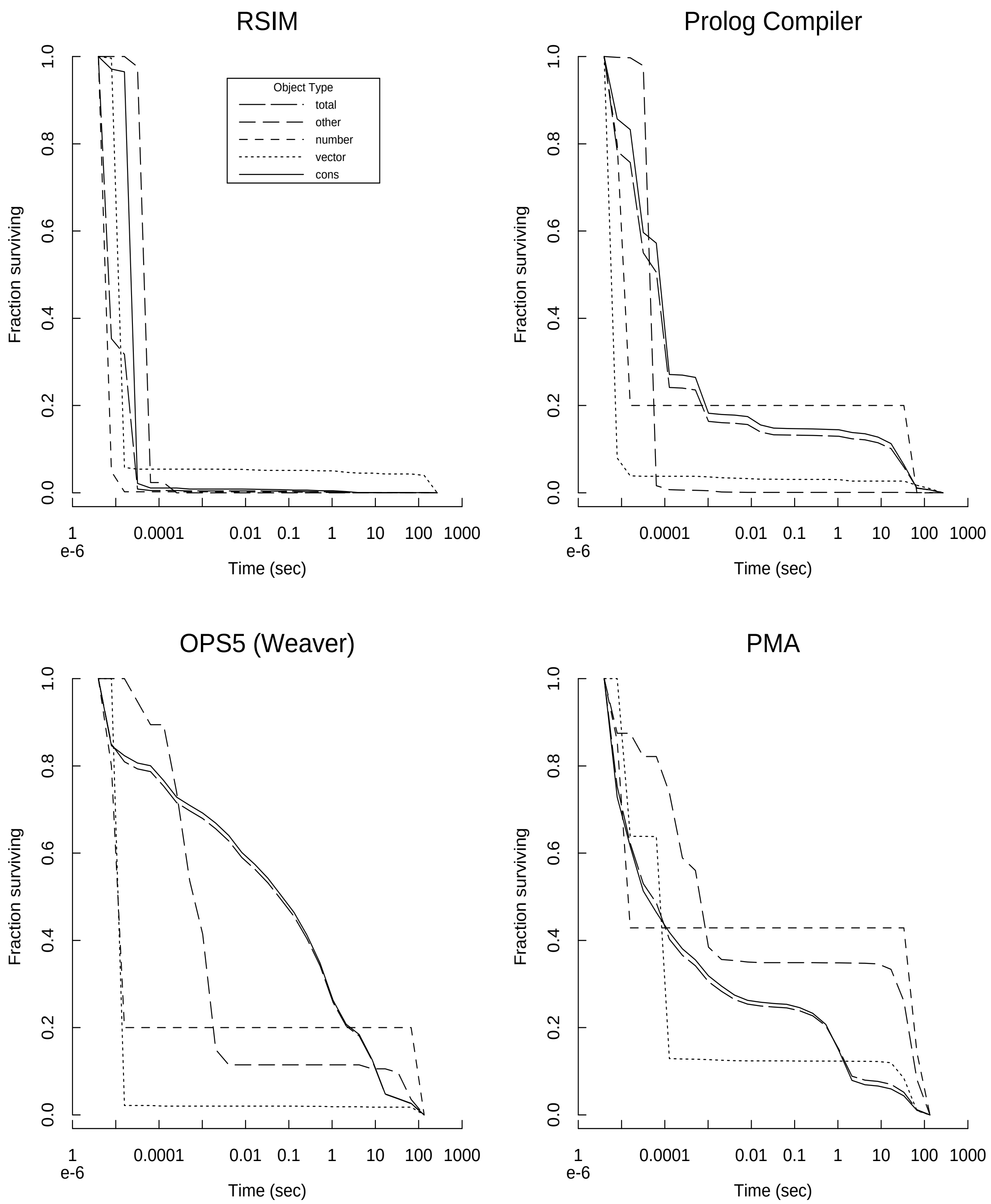

Figure C.18: Survival Distribution of Objects Referenced by Object Type.

Age	OPS5 (Weaver) Fraction surviving					PMA Fraction surviving				
(sec)	cons	vector	number	other	total	cons	vector	number	other	total
0.000004	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.000008	0.85	1.00	0.80	1.00	0.85	0.73	1.00	0.86	0.88	0.75
0.000016	0.82	0.02	0.20	1.00	0.81	0.62	0.64	0.43	0.88	0.62
0.000032	0.81	0.02	0.20	0.95	0.79	0.51	0.64	0.43	0.82	0.53
0.000064	0.80	0.02	0.20	0.89	0.79	0.46	0.64	0.43	0.82	0.49
0.000128	0.77	0.02	0.20	0.89	0.75	0.42	0.13	0.43	0.74	0.40
0.000256	0.73	0.02	0.20	0.74	0.72	0.38	0.13	0.43	0.59	0.37
0.000512	0.71	0.02	0.20	0.54	0.70	0.36	0.13	0.43	0.56	0.34
0.001024	0.69	0.02	0.20	0.41	0.68	0.32	0.13	0.43	0.38	0.31
0.002048	0.67	0.02	0.20	0.15	0.66	0.29	0.13	0.43	0.36	0.28
0.004096	0.64	0.02	0.20	0.11	0.63	0.27	0.12	0.43	0.35	0.26
0.008192	0.60	0.02	0.20	0.11	0.59	0.26	0.12	0.43	0.35	0.25
0.016384	0.57	0.02	0.20	0.11	0.56	0.26	0.12	0.43	0.35	0.25
0.032768	0.54	0.02	0.20	0.11	0.53	0.26	0.12	0.43	0.35	0.25
0.065536	0.50	0.02	0.20	0.11	0.49	0.25	0.12	0.43	0.35	0.25
0.131072	0.46	0.02	0.20	0.11	0.46	0.25	0.12	0.43	0.35	0.24
0.262144	0.41	0.02	0.20	0.11	0.40	0.23	0.12	0.43	0.35	0.23
0.524288	0.35	0.02	0.20	0.11	0.34	0.21	0.12	0.43	0.35	0.20
1.048580	0.26	0.02	0.20	0.11	0.26	0.15	0.12	0.43	0.35	0.15
2.097150	0.21	0.02	0.20	0.11	0.20	0.08	0.12	0.43	0.35	0.09
4.194300	0.18	0.02	0.20	0.11	0.18	0.07	0.12	0.43	0.35	0.08
8.388610	0.13	0.02	0.20	0.11	0.12	0.07	0.12	0.43	0.35	0.08
16.777200	0.05	0.02	0.20	0.11	0.05	0.06	0.12	0.43	0.33	0.07
33.554400	0.04	0.02	0.20	0.10	0.04	0.04	0.08	0.43	0.26	0.05
67.108900	0.03	0.02	0.20	0.04	0.03	0.01	0.01	0.14	0.08	0.01
134.218000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table C.23: Survival Distribution of Objects Referenced by Object Type.

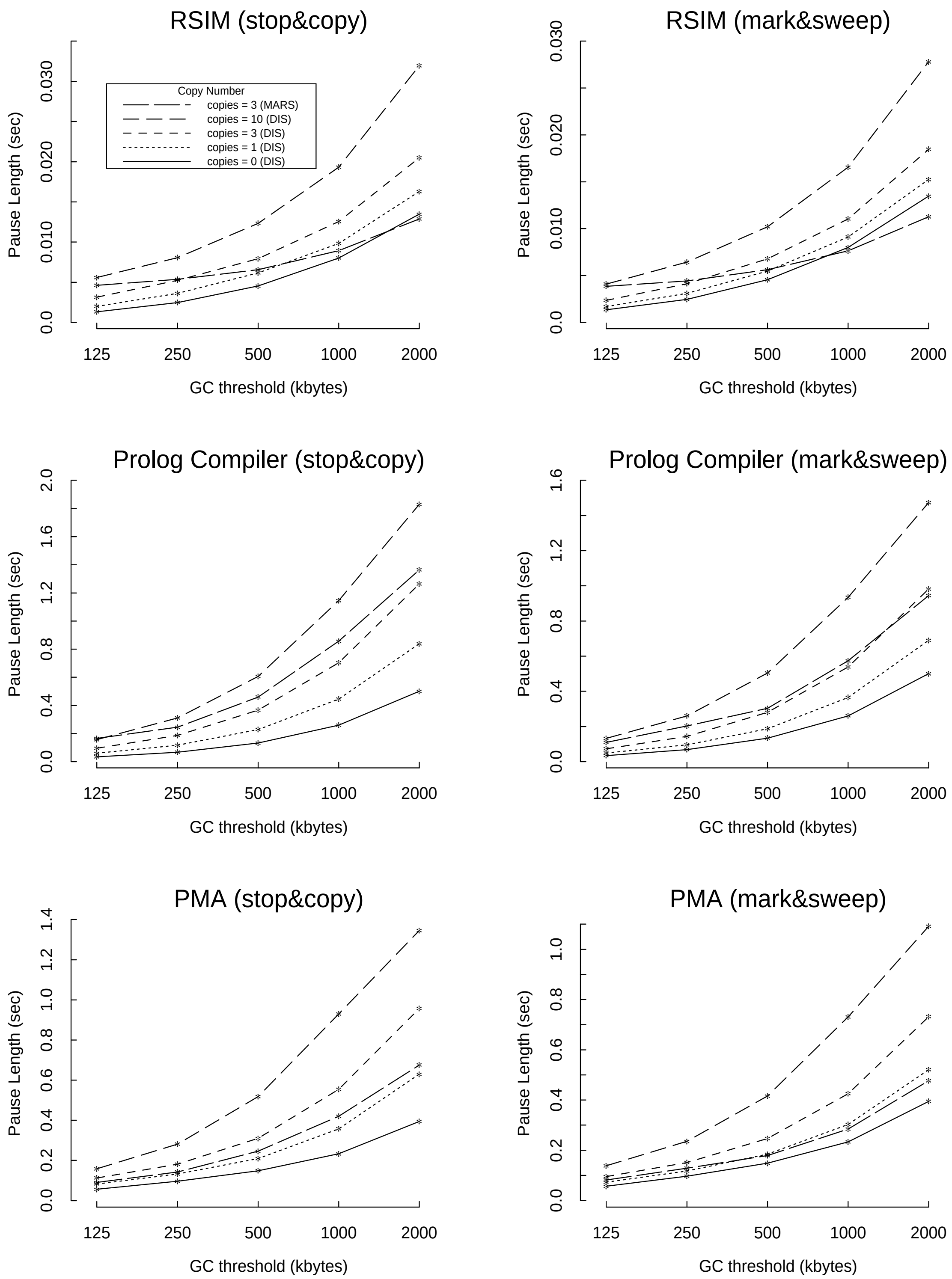

Figure C.19: Pause Lengths for Three Applications

RSIM (stop&copy)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.001	0.002	0.005	0.008	0.013
copies = 1 (DIS)	0.002	0.004	0.006	0.010	0.016
copies = 3 (DIS)	0.003	0.005	0.008	0.013	0.020
copies = 10 (DIS)	0.006	0.008	0.012	0.019	0.032
copies = 3 (MARS)	0.005	0.005	0.007	0.009	0.013

RSIM (mark&sweep)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.001	0.002	0.005	0.008	0.013
copies = 1 (DIS)	0.002	0.003	0.005	0.009	0.015
copies = 3 (DIS)	0.002	0.004	0.007	0.011	0.018
copies = 10 (DIS)	0.004	0.006	0.010	0.017	0.028
copies = 3 (MARS)	0.004	0.004	0.006	0.008	0.011

Prolog Compiler (stop&copy)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.034	0.068	0.133	0.260	0.501
copies = 1 (DIS)	0.060	0.117	0.229	0.445	0.839
copies = 3 (DIS)	0.096	0.187	0.366	0.704	1.263
copies = 10 (DIS)	0.158	0.310	0.606	1.145	1.830
copies = 3 (MARS)	0.166	0.246	0.459	0.857	1.364

Prolog Compiler (mark&sweep)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.034	0.068	0.133	0.260	0.501
copies = 1 (DIS)	0.049	0.096	0.187	0.365	0.690
copies = 3 (DIS)	0.073	0.144	0.281	0.539	0.982
copies = 10 (DIS)	0.133	0.260	0.505	0.935	1.474
copies = 3 (MARS)	0.110	0.203	0.303	0.574	0.944

PMA (stop&copy)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.057	0.097	0.148	0.233	0.394
copies = 1 (DIS)	0.083	0.133	0.209	0.357	0.630
copies = 3 (DIS)	0.113	0.182	0.311	0.554	0.957
copies = 10 (DIS)	0.159	0.282	0.518	0.930	1.345
copies = 3 (MARS)	0.090	0.143	0.246	0.420	0.676

PMA (mark&sweep)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.057	0.097	0.148	0.233	0.394
copies = 1 (DIS)	0.073	0.118	0.184	0.303	0.521
copies = 3 (DIS)	0.095	0.151	0.247	0.425	0.732
copies = 10 (DIS)	0.138	0.236	0.416	0.730	1.092
copies = 3 (MARS)	0.082	0.128	0.179	0.284	0.477

Table C.24: Pause Lengths for Three Applications

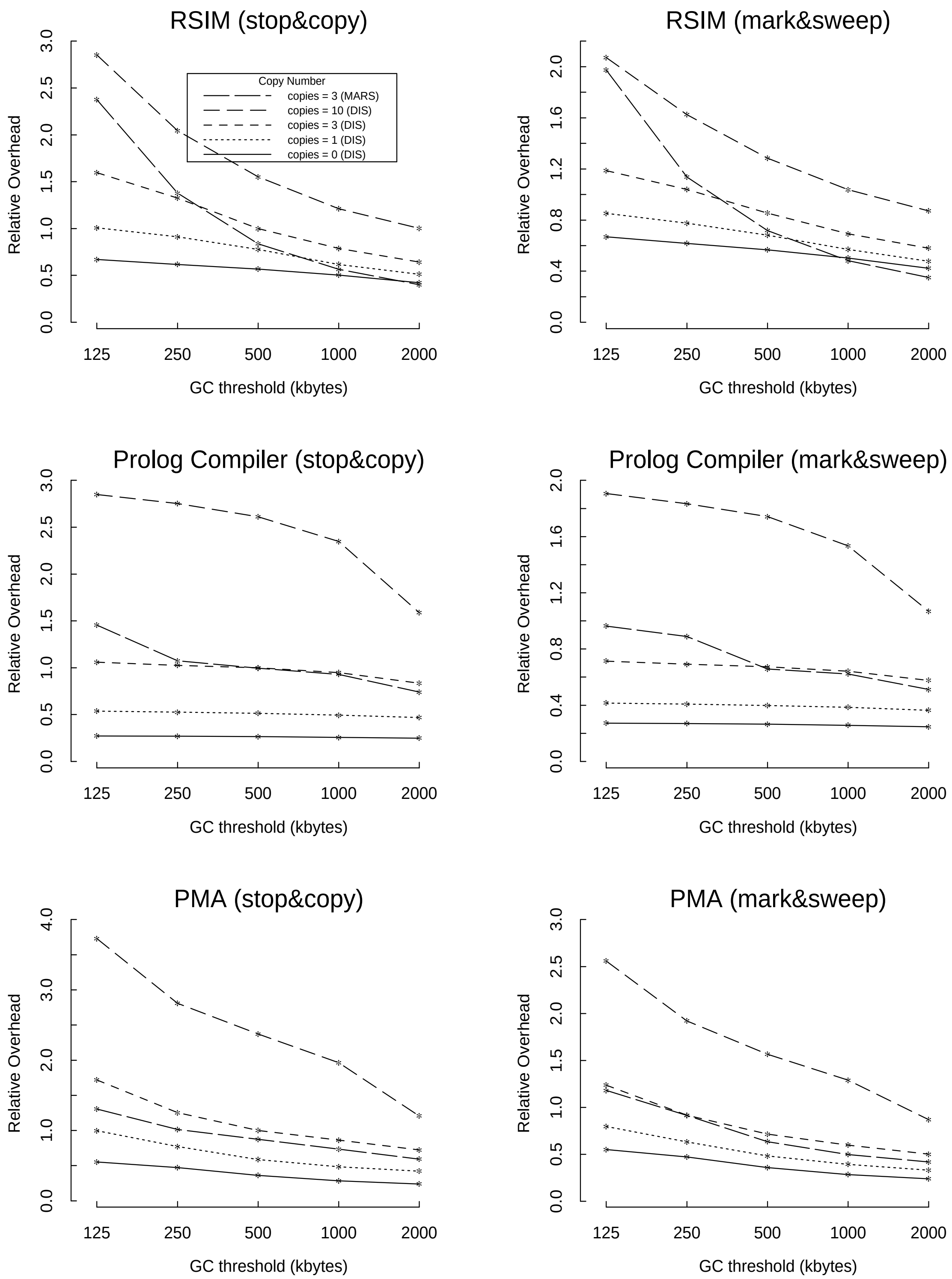

Figure C.20: Relative CPU Overhead for Three Applications

RSIM (stop&copy)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.669	0.617	0.568	0.502	0.422
copies = 1 (DIS)	1.008	0.911	0.777	0.616	0.511
copies = 3 (DIS)	1.596	1.325	1.000	0.788	0.642
copies = 10 (DIS)	2.854	2.045	1.552	1.211	1.001
copies = 3 (MARS)	2.376	1.378	0.837	0.564	0.400

RSIM (mark&sweep)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.669	0.617	0.568	0.502	0.422
copies = 1 (DIS)	0.853	0.775	0.682	0.571	0.478
copies = 3 (DIS)	1.186	1.039	0.854	0.692	0.579
copies = 10 (DIS)	2.072	1.624	1.284	1.038	0.872
copies = 3 (MARS)	1.975	1.136	0.717	0.482	0.350

Prolog Compiler (stop&copy)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.272	0.269	0.265	0.256	0.247
copies = 1 (DIS)	0.537	0.527	0.513	0.495	0.468
copies = 3 (DIS)	1.059	1.026	1.000	0.948	0.835
copies = 10 (DIS)	2.850	2.753	2.613	2.348	1.589
copies = 3 (MARS)	1.456	1.075	0.996	0.930	0.740

Prolog Compiler (mark&sweep)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.272	0.269	0.265	0.256	0.247
copies = 1 (DIS)	0.415	0.408	0.398	0.385	0.364
copies = 3 (DIS)	0.714	0.692	0.674	0.642	0.577
copies = 10 (DIS)	1.906	1.834	1.743	1.534	1.069
copies = 3 (MARS)	0.962	0.888	0.658	0.622	0.512

PMA (stop&copy)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.552	0.472	0.360	0.285	0.239
copies = 1 (DIS)	0.996	0.772	0.586	0.485	0.421
copies = 3 (DIS)	1.720	1.254	1.000	0.862	0.722
copies = 10 (DIS)	3.730	2.808	2.372	1.964	1.206
copies = 3 (MARS)	1.305	1.016	0.874	0.735	0.592

PMA (mark&sweep)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.552	0.472	0.360	0.285	0.239
copies = 1 (DIS)	0.796	0.634	0.484	0.392	0.332
copies = 3 (DIS)	1.238	0.917	0.716	0.598	0.502
copies = 10 (DIS)	2.559	1.920	1.566	1.290	0.871
copies = 3 (MARS)	1.182	0.916	0.634	0.498	0.418

Table C.25: Relative CPU Overhead for Three Applications

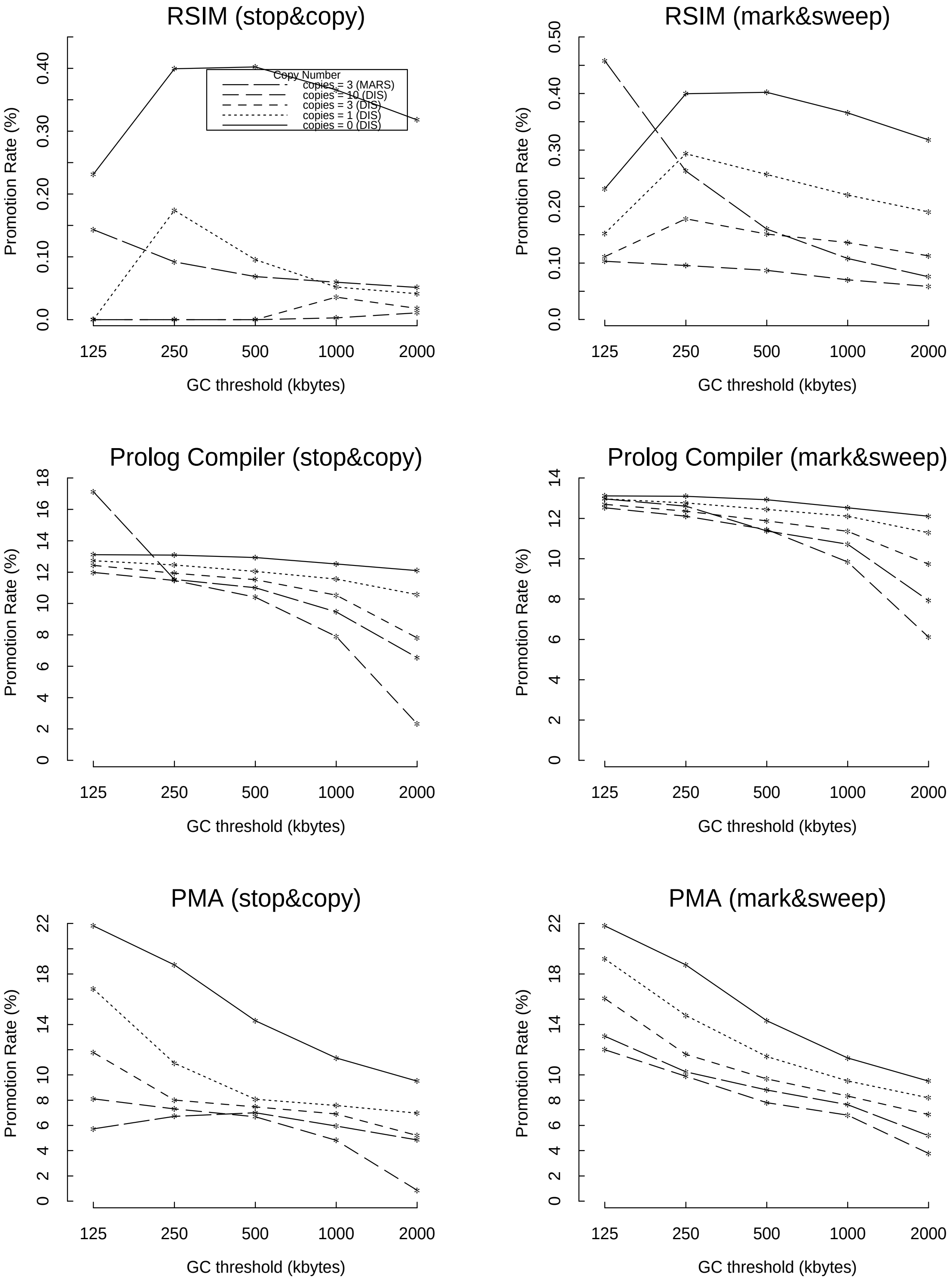

Figure C.21: Promotion Rates for Three Applications

RSIM (stop&copy)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.231	0.400	0.402	0.366	0.318
copies = 1 (DIS)	0.000	0.174	0.096	0.052	0.041
copies = 3 (DIS)	0.000	0.000	0.000	0.036	0.018
copies = 10 (DIS)	0.000	0.000	0.000	0.003	0.011
copies = 3 (MARS)	0.143	0.092	0.069	0.060	0.051

RSIM (mark&sweep)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.231	0.400	0.402	0.366	0.318
copies = 1 (DIS)	0.152	0.294	0.257	0.220	0.190
copies = 3 (DIS)	0.111	0.178	0.151	0.136	0.113
copies = 10 (DIS)	0.103	0.096	0.087	0.070	0.059
copies = 3 (MARS)	0.458	0.263	0.160	0.108	0.075

Prolog Compiler (stop&copy)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	13.114	13.093	12.928	12.523	12.103
copies = 1 (DIS)	12.728	12.454	12.040	11.556	10.566
copies = 3 (DIS)	12.433	11.925	11.527	10.526	7.803
copies = 10 (DIS)	11.978	11.471	10.400	7.903	2.321
copies = 3 (MARS)	17.131	11.527	11.013	9.464	6.548

Prolog Compiler (mark&sweep)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	13.114	13.093	12.928	12.523	12.103
copies = 1 (DIS)	12.971	12.763	12.446	12.105	11.286
copies = 3 (DIS)	12.701	12.357	11.882	11.363	9.732
copies = 10 (DIS)	12.523	12.110	11.448	9.837	6.116
copies = 3 (MARS)	12.971	12.602	11.382	10.716	7.921

PMA (stop&copy)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	21.814	18.713	14.302	11.348	9.506
copies = 1 (DIS)	16.811	10.929	8.073	7.579	6.982
copies = 3 (DIS)	11.770	8.006	7.474	6.917	5.206
copies = 10 (DIS)	8.113	7.309	6.695	4.815	0.871
copies = 3 (MARS)	5.735	6.724	7.002	5.950	4.858

PMA (mark&sweep)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	21.814	18.713	14.302	11.348	9.506
copies = 1 (DIS)	19.194	14.706	11.454	9.535	8.185
copies = 3 (DIS)	16.056	11.639	9.693	8.328	6.862
copies = 10 (DIS)	11.996	9.888	7.809	6.807	3.785
copies = 3 (MARS)	13.077	10.249	8.814	7.640	5.181

Table C.26: Promotion Rates for Three Applications

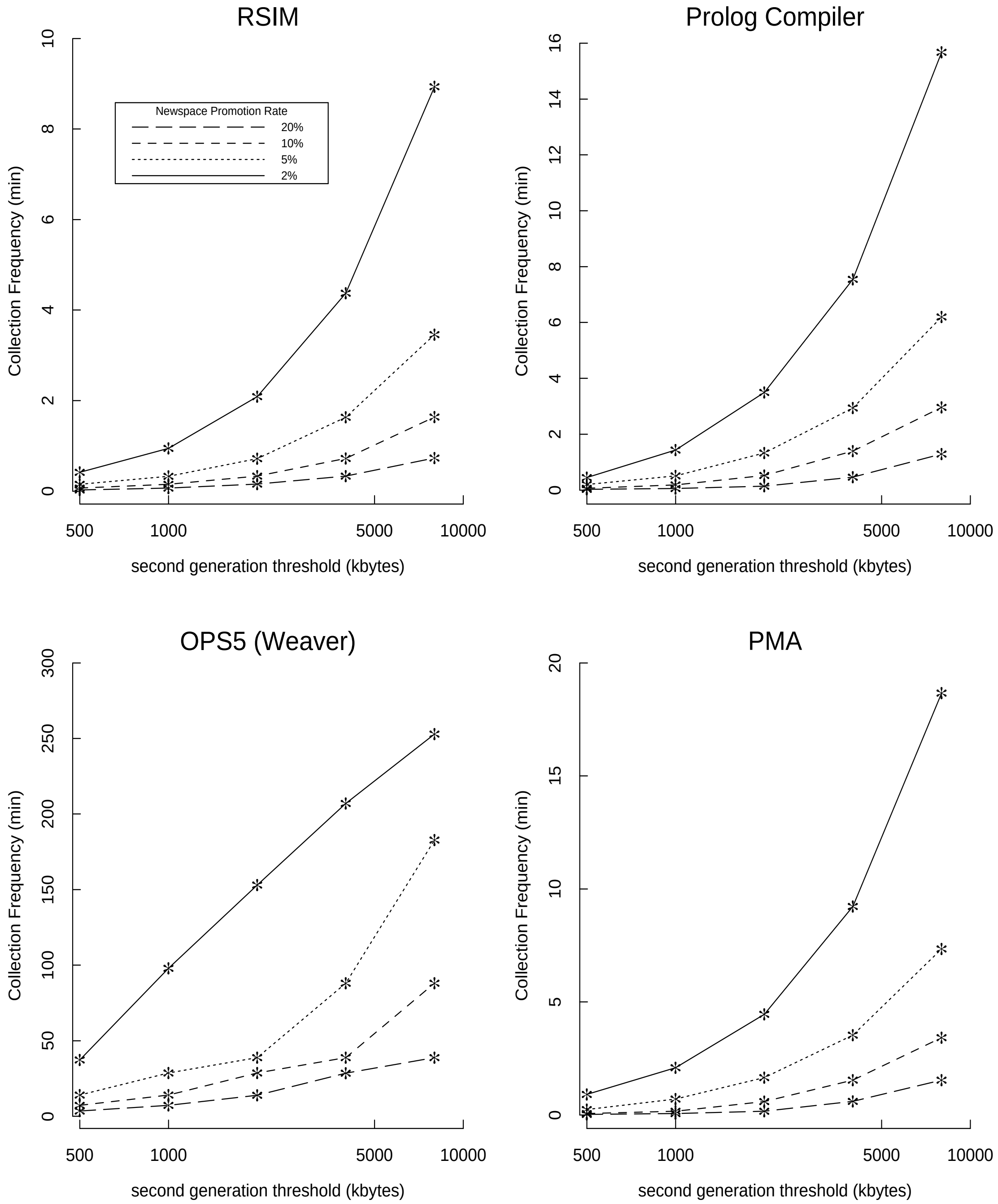

Figure C.22: Second Generation Collection Frequencies for Four Applications

RSIM

Newspace	Collection Frequency (min)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.417	0.946	2.089	4.374	8.936
5%	0.153	0.328	0.720	1.635	3.458
10%	0.067	0.154	0.331	0.725	1.640
20%	0.029	0.068	0.156	0.335	0.734

Prolog Compiler

Newspace	Collection Frequency (min)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.452	1.435	3.500	7.550	15.673
5%	0.200	0.510	1.328	2.943	6.201
10%	0.064	0.185	0.526	1.400	2.961
20%	0.027	0.057	0.144	0.468	1.289

OPS5 (Weaver)

Newspace	Collection Frequency (min)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	37.225	97.918	152.986	207.033	252.927
5%	14.024	28.728	38.761	88.120	182.760
10%	7.211	14.001	28.728	38.761	88.120
20%	3.491	7.174	13.941	28.564	38.761

PMA

Newspace	Collection Frequency (min)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.915	2.094	4.454	9.228	18.661
5%	0.242	0.710	1.653	3.534	7.352
10%	0.068	0.169	0.600	1.541	3.422
20%	0.032	0.068	0.171	0.602	1.544

Table C.27: Second Generation Collection Frequencies for Four Applications

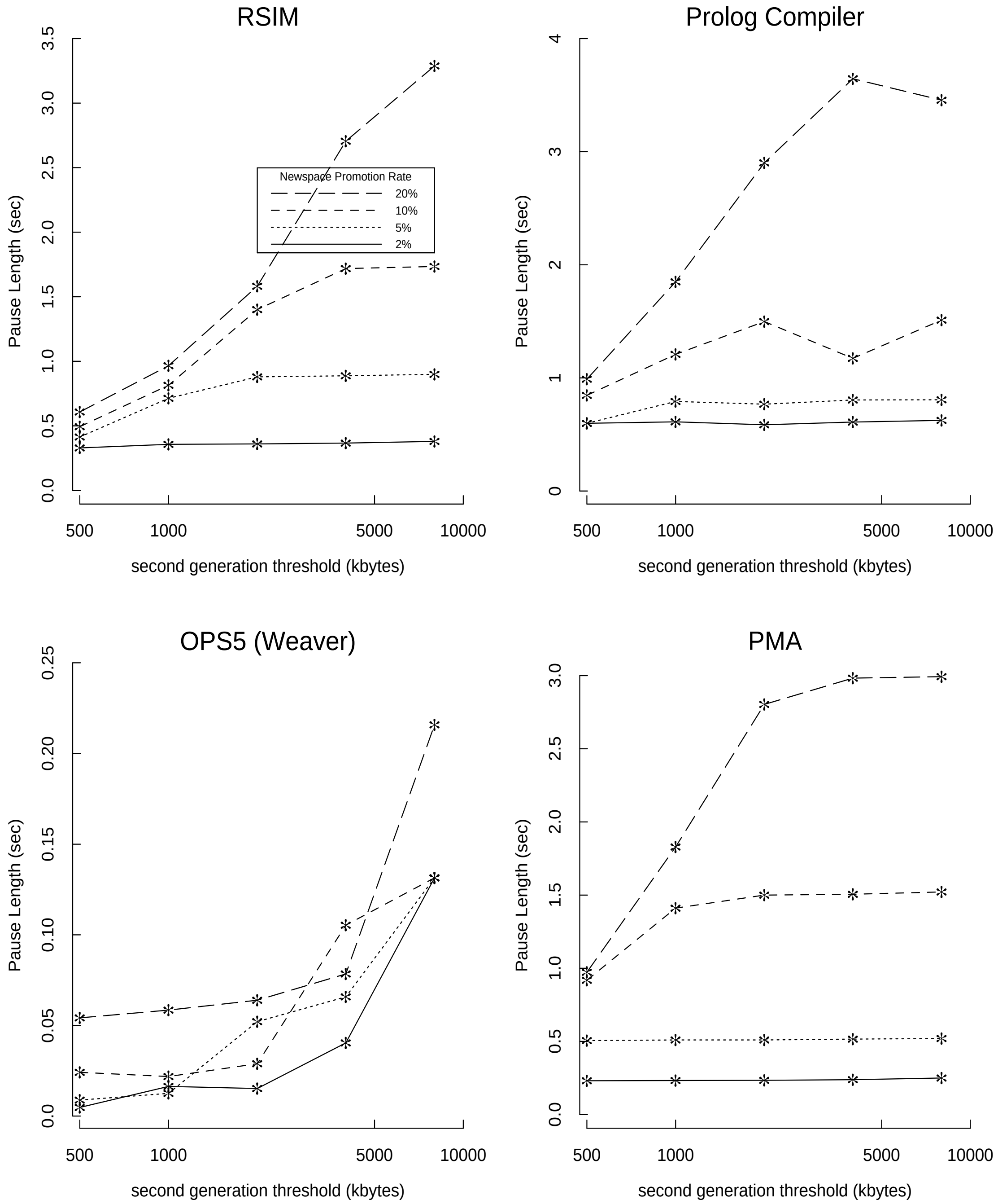

Figure C.23: Second Generation Pause Lengths for Four Applications

RSIM

Newspace	Pause Length (sec)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.330	0.358	0.361	0.368	0.380
5%	0.414	0.714	0.881	0.888	0.900
10%	0.495	0.815	1.401	1.718	1.735
20%	0.609	0.966	1.581	2.704	3.287

Prolog Compiler

Newspace	Pause Length (sec)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.598	0.610	0.585	0.610	0.623
5%	0.597	0.791	0.766	0.803	0.807
10%	0.844	1.208	1.497	1.171	1.511
20%	0.988	1.849	2.901	3.645	3.454

OPS5 (Weaver)

Newspace	Pause Length (sec)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.005	0.016	0.015	0.040	0.131
5%	0.009	0.013	0.052	0.066	0.131
10%	0.024	0.022	0.029	0.105	0.131
20%	0.054	0.058	0.064	0.078	0.216

PMA

Newspace	Pause Length (sec)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.231	0.232	0.234	0.238	0.250
5%	0.506	0.509	0.510	0.515	0.520
10%	0.918	1.412	1.500	1.506	1.522
20%	0.973	1.829	2.804	2.984	2.993

Table C.28: Second Generation Pause Lengths for Four Applications

Prolog Compiler (stop&copy) | **PMA (stop&copy)**

Threshold (kbytes)	Pause Length (msec)			Pause Length (msec)		
	copies=0	copies=3	copies=10	copies=0	copies=3	copies=10
2000	5.013	12.632	18.297	3.941	9.573	13.452
4000	9.340	18.523	21.913	6.858	13.367	14.457
8000	15.317	21.626	23.266	11.189	14.647	14.853
16000	19.862	23.877	24.250	14.262	15.053	15.428
32000	23.033	24.629	25.434	15.436	15.819	16.586

Prolog Compiler (stop&copy) | **PMA (stop&copy)**

Threshold (kbytes)	Collection Frequency (sec)			Collection Frequency (sec)		
	copies=0	copies=3	copies=10	copies=0	copies=3	copies=10
2000	0.048	0.036	0.028	0.056	0.046	0.038
4000	0.097	0.078	0.072	0.113	0.096	0.093
8000	0.195	0.171	0.168	0.226	0.206	0.206
16000	0.391	0.363	0.363	0.452	0.431	0.432
32000	0.782	0.753	0.754	0.905	0.882	0.884

Prolog Compiler (stop&copy) | **PMA (stop&copy)**

Threshold (kbytes)	Promotion Rate (%)			Promotion Rate (%)		
	copies=0	copies=3	copies=10	copies=0	copies=3	copies=10
2000	12.103	7.803	2.321	9.506	5.206	0.871
4000	11.351	2.841	0.532	8.274	1.374	0.000
8000	9.247	0.733	0.000	6.755	0.024	0.000
16000	6.000	0.086	0.000	4.307	0.000	0.000
32000	3.482	0.000	0.000	2.333	0.000	0.000

Table C.29: Predicted Performance for Two Applications. Predicted CPU speed is 100 times a Sun4/280.

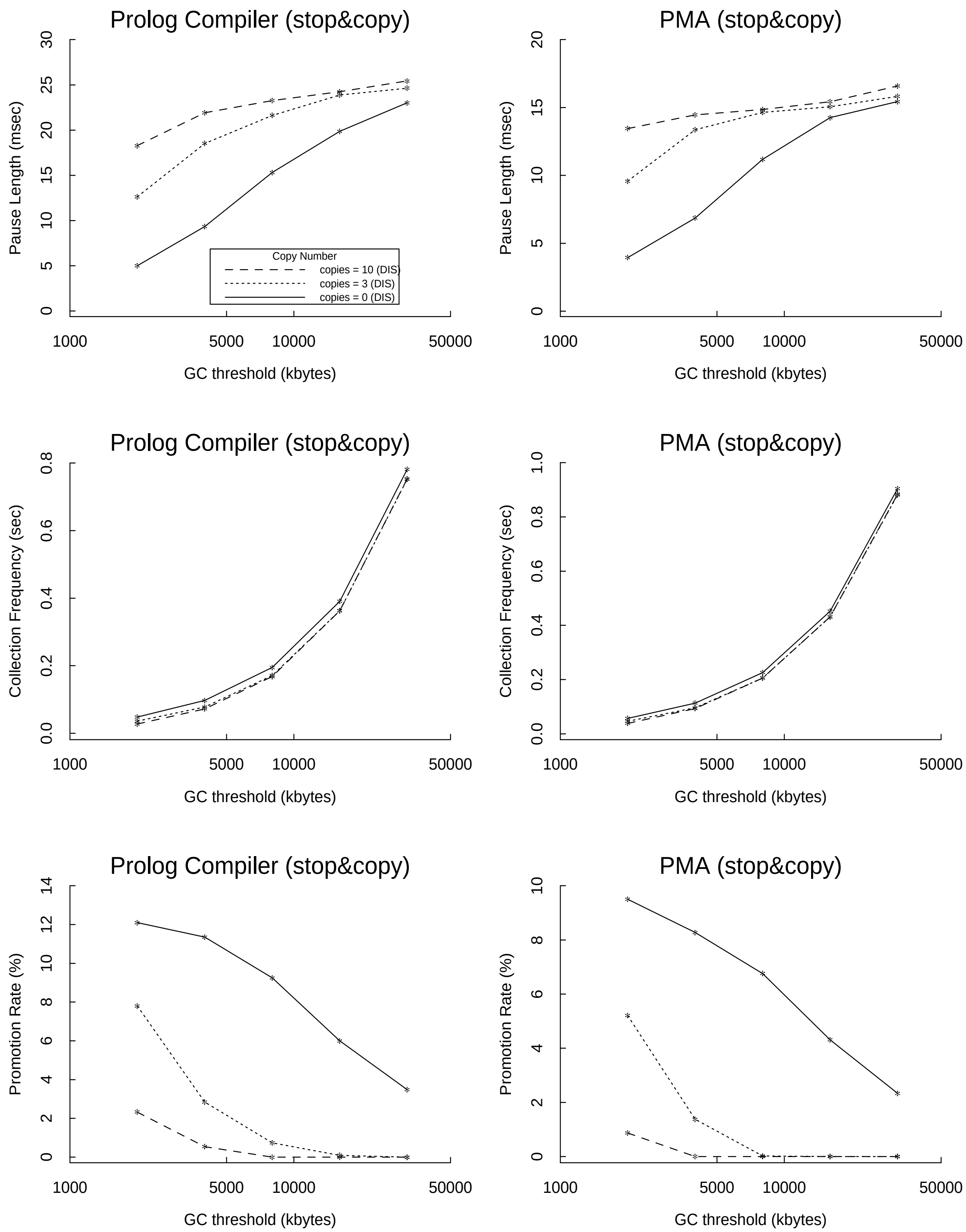

Figure C.24: Predicted Performance for Two Applications. Predicted CPU speed is 100 times a Sun4/280.

Prolog Compiler (stop&copy)

Threshold (kbytes)	Pause Length (msec)		
	copies=0	copies=3	copies=10
2000	5.703	15.577	25.556
4000	11.168	30.918	50.892
8000	22.060	61.432	101.018
16000	43.677	120.920	199.638
32000	86.377	237.526	393.544
64000	168.614	462.801	764.330
128000	328.515	877.463	1379.140

PMA (stop&copy)

Pause Length (msec)		
copies=0	copies=3	copies=10
10.219	22.694	31.428
19.910	43.764	60.855
38.081	79.759	111.269
69.157	131.563	192.502
113.114	216.873	347.934
171.256	379.520	640.750
276.772	677.312	1107.760

Prolog Compiler (stop&copy)

Threshold (kbytes)	Collection Frequency (sec)		
	copies=0	copies=3	copies=10
2000	0.048	0.035	0.021
4000	0.097	0.070	0.043
8000	0.195	0.140	0.086
16000	0.391	0.282	0.175
32000	0.782	0.568	0.355
64000	1.567	1.145	0.734
128000	3.143	2.328	1.602

PMA (stop&copy)

Collection Frequency (sec)		
copies=0	copies=3	copies=10
0.056	0.033	0.017
0.113	0.067	0.036
0.226	0.138	0.082
0.452	0.299	0.204
0.905	0.658	0.462
1.813	1.394	0.995
3.625	2.889	2.186

Prolog Compiler (stop&copy)

Threshold (kbytes)	Promotion Rate (%)		
	copies=0	copies=3	copies=10
2000	13.770	12.822	12.488
4000	13.574	12.674	12.376
8000	13.319	12.648	11.934
16000	13.195	12.127	11.614
32000	13.060	11.839	11.056
64000	12.776	11.232	9.855
128000	12.487	9.950	5.660

PMA (stop&copy)

Promotion Rate (%)		
copies=0	copies=3	copies=10
24.666	22.509	21.220
24.033	20.451	17.553
22.998	15.532	10.709
20.892	9.080	7.604
17.101	7.711	7.165
12.968	7.372	6.198
10.484	6.439	3.514

Table C.30: Predicted Performance for Two Applications. Predicted lifespans are 100 times those measured. Predicted CPU speed is 100 times a Sun4/280.

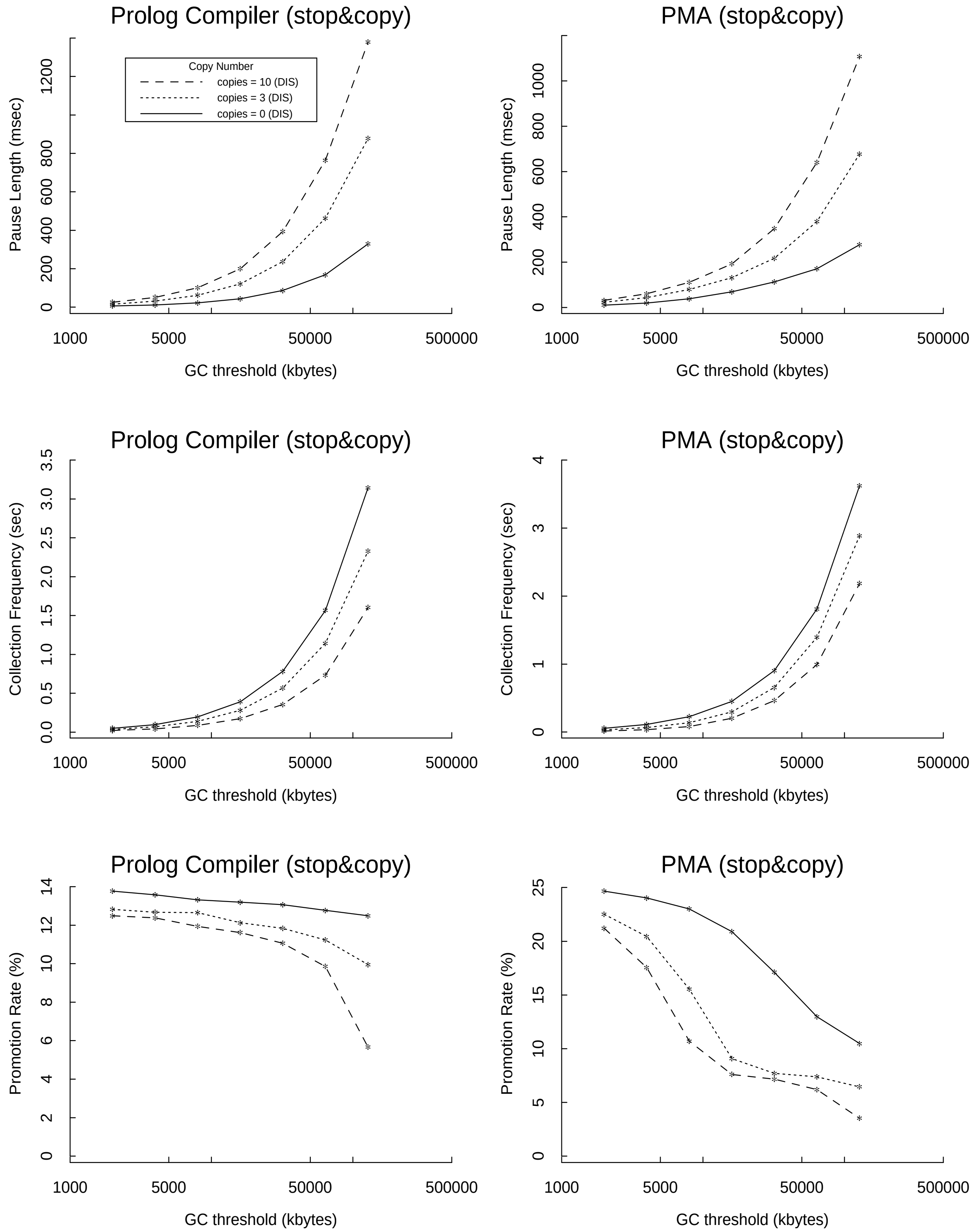

Figure C.25: Predicted Performance for Two Applications. Predicted lifespans are 100 times those measured. Predicted CPU speed is 100 times a Sun4/280.

Prolog Compiler (stop&copy)

Threshold (kbytes)	Pause Length (msec) pr=10%	pr=20%	pr=30%	pr=40%
2000	40.89	40.47	40.45	40.29
4000	81.34	80.80	80.74	80.44
8000	160.77	160.24	160.95	160.59
16000	313.41	318.09	318.61	318.36
32000	586.57	628.39	632.62	632.06
64000	991.50	1221.58	1242.62	1248.08
128000	1302.66	2153.66	2359.62	2424.22
256000	1399.07	2906.13	3917.18	4256.31
512000	1542.27	3144.83	4659.93	5698.56
1024000	619.61	3471.89	4064.81	6147.95

PMA (stop&copy)

Pause Length (msec) pr=10%	pr=20%	pr=30%	pr=40%
40.53	38.03	39.73	40.06
80.12	70.67	76.05	77.91
159.35	129.28	136.64	143.94
315.37	245.75	241.19	251.33
613.64	470.79	449.58	461.54
1095.83	867.38	836.13	864.54
1398.02	1237.38	1411.31	1556.20
1316.70	1299.52	1590.83	2104.88
1200.22	1232.74	1562.99	2173.17
2260.20	1129.83	2268.29	2042.22

Prolog Compiler (stop&copy)

Threshold (kbytes)	Collection Frequency (sec) pr=10%	pr=20%	pr=30%	pr=40%
2000	0.12	0.06	0.04	0.03
4000	0.25	0.13	0.08	0.06
8000	0.50	0.26	0.17	0.13
16000	1.04	0.52	0.35	0.26
32000	2.22	1.06	0.70	0.53
64000	5.49	2.19	1.45	1.08
128000	17.56	4.98	3.05	2.23
256000	48.89	15.54	7.08	5.08
512000	103.47	46.75	25.20	15.85
1024000	600.07	101.40	82.81	47.19

PMA (stop&copy)

Collection Frequency (sec) pr=10%	pr=20%	pr=30%	pr=40%
0.14	0.08	0.05	0.04
0.29	0.17	0.10	0.08
0.60	0.41	0.23	0.16
1.23	0.93	0.59	0.41
2.50	2.01	1.38	0.98
5.86	4.22	3.00	2.18
18.34	10.67	6.60	4.65
61.44	29.02	18.42	11.86
160.66	72.30	45.66	30.61
172.95	171.49	66.66	73.92

Prolog Compiler (stop&copy)

Threshold (kbytes)	Promotion Rate (%) pr=10%	pr=20%	pr=30%	pr=40%
2000	98.06	94.20	93.86	92.34
4000	95.91	93.39	93.04	91.84
8000	91.34	90.37	91.97	91.05
16000	82.32	87.71	88.13	88.01
32000	61.97	65.64	65.62	65.64
64000	28.20	32.69	33.07	32.82
128000	6.18	16.72	16.53	16.35
256000	0.00	5.89	8.06	8.36
512000	0.00	0.00	3.24	4.26
1024000	0.00	0.00	0.00	0.00

PMA (stop&copy)

Promotion Rate (%) pr=10%	pr=20%	pr=30%	pr=40%
94.52	72.79	87.20	90.23
90.08	53.62	72.65	80.38
88.23	40.70	45.60	57.10
84.62	38.13	31.92	33.80
65.64	35.18	30.04	30.81
34.83	27.34	25.94	28.14
1.36	5.14	14.58	16.41
0.00	0.00	0.54	3.52
0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00

Table C.31: Second Generation Metrics for Four Applications Assuming Longer Running Programs and Faster CPU's. Object lifespan is assumed to be 100 times the lifespan actually observed.

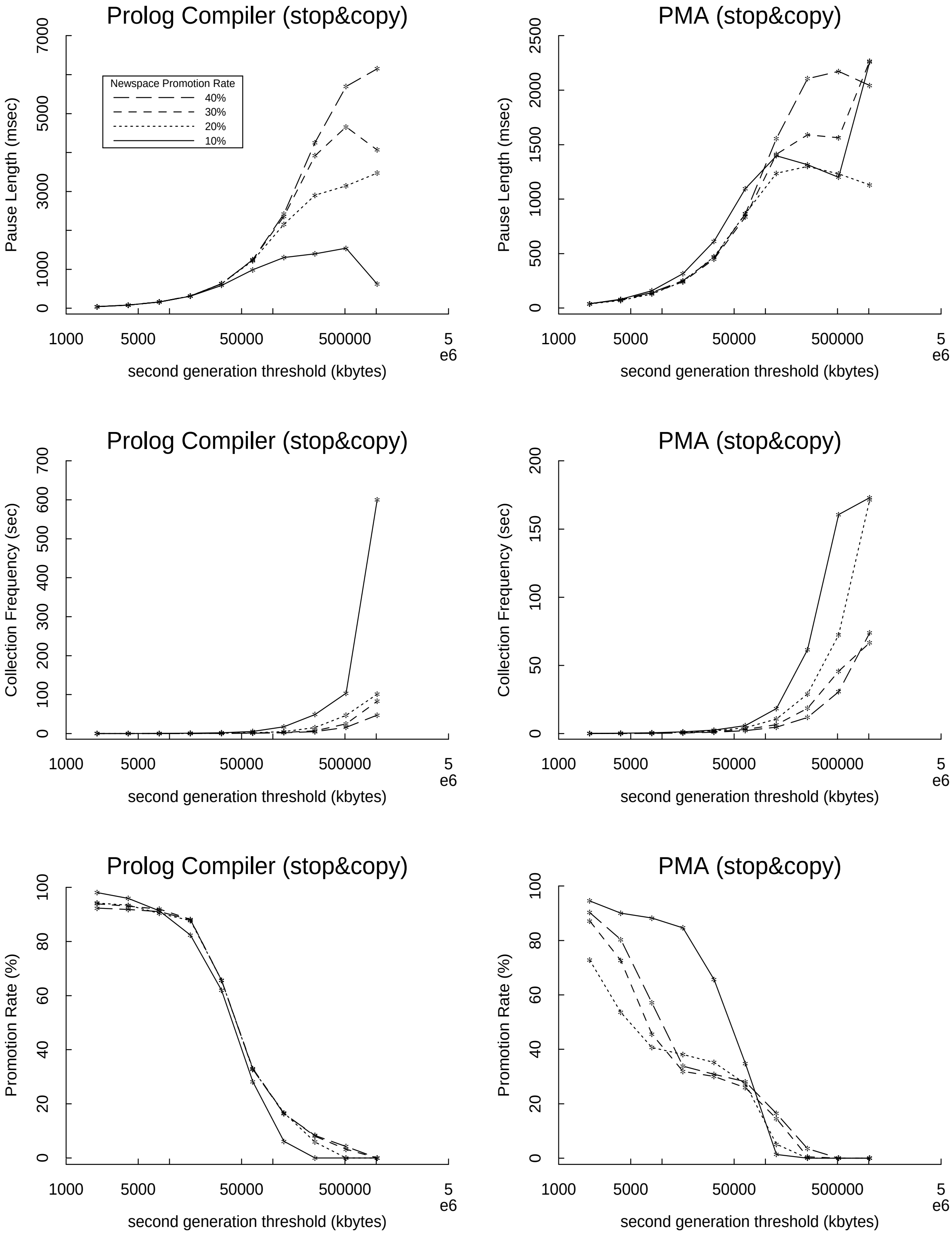

Figure C.26: Second Generation Metrics for Four Applications Assuming Longer Running Programs and Faster CPU's. Object lifespan is assumed to be 100 times the lifespan actually observed.

Prolog Compiler (stop&copy)

Threshold Size (kbytes)	Pause Length (msec) pr = 2%	pr = 4%	pr = 6%	pr = 8%
2000	39.64	40.46	40.47	40.69
4000	75.88	79.69	79.72	80.65
8000	131.86	153.77	155.79	158.23
16000	207.45	277.18	297.01	302.50
32000	312.39	421.67	505.77	557.56
64000	269.69	621.83	723.49	836.81
128000	226.30	539.95	874.67	1166.14
256000	182.92	451.67	758.92	1001.79
512000	293.55	365.21	912.76	837.54
1024000	0.00	586.08	1012.72	681.44

PMA (stop&copy)

Pause Length (msec) pr = 2%	pr = 4%	pr = 6%	pr = 8%
39.17	40.38	40.72	40.93
73.44	79.32	80.80	81.63
122.84	151.90	158.80	162.25
181.13	265.13	302.93	320.00
163.12	392.48	529.37	612.81
161.02	360.98	622.66	1059.41
182.69	356.32	740.60	1127.14
102.58	403.97	550.72	1108.26
106.39	226.41	704.79	1254.52
114.00	234.00	394.83	698.87

Prolog Compiler (stop&copy)

Threshold Size (kbytes)	Collection Frequency (sec) pr = 2%	pr = 4%	pr = 6%	pr = 8%
2000	0.64	0.31	0.21	0.16
4000	1.35	0.64	0.42	0.31
8000	3.08	1.33	0.88	0.64
16000	8.72	2.95	1.87	1.36
32000	21.41	8.34	4.34	3.02
64000	72.97	21.38	13.48	8.07
128000	201.26	72.97	36.00	22.12
256000	539.36	201.26	106.70	74.21
512000	675.10	539.36	190.30	202.91
1024000	0.00	675.10	362.49	539.36

PMA (stop&copy)

Collection Frequency (sec) pr = 2%	pr = 4%	pr = 6%	pr = 8%
0.75	0.36	0.24	0.18
1.63	0.74	0.48	0.36
4.02	1.57	0.99	0.72
10.66	3.56	2.13	1.48
37.25	9.97	4.59	3.07
88.54	36.00	17.51	6.80
166.93	87.19	40.21	28.40
636.49	165.57	135.70	79.72
1285.51	635.59	222.56	157.89
2585.34	1283.72	849.84	628.43

Table C.32: Third Generation Metrics for Four Applications Assuming Longer Running Programs and Faster CPU's. Object lifespan is assumed to be 100 times the lifespan actually observed.

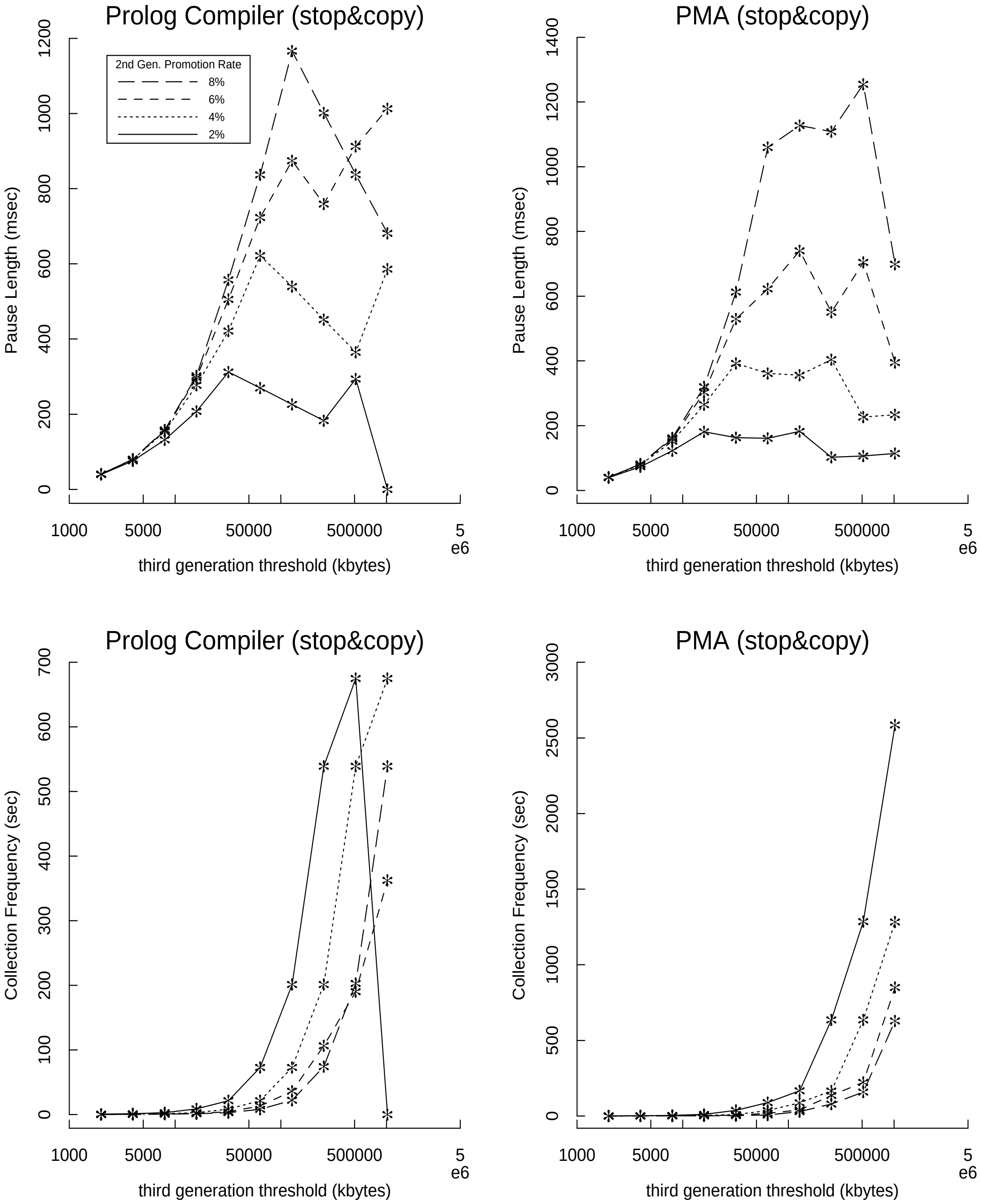

Figure C.27: Third Generation Metrics for Four Applications Assuming Longer Running Programs and Faster CPU's. Object lifespan is assumed to be 100 times the lifespan actually observed.

RSIM

Algorithm (Cache Size)	Total miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (512k)	0.451	0.555	0.700	0.708	0.708
mark-and-sweep (512k)	0.232	0.160	0.158	0.276	0.387
stop-and-copy (1M)	0.419	0.522	0.579	0.646	0.655
mark-and-sweep (1M)	0.047	0.047	0.051	0.080	0.221
stop-and-copy (2M)	0.165	0.099	0.080	0.121	0.620
mark-and-sweep (2M)	0.032	0.033	0.036	0.042	0.056

Prolog Compiler

Algorithm (Cache Size)	Total miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (512k)	0.391	0.450	0.560	0.591	0.623
mark-and-sweep (512k)	0.300	0.269	0.407	0.571	0.577
stop-and-copy (1M)	0.307	0.386	0.435	0.516	0.545
mark-and-sweep (1M)	0.232	0.177	0.219	0.340	0.494
stop-and-copy (2M)	0.116	0.130	0.157	0.276	0.472
mark-and-sweep (2M)	0.182	0.139	0.156	0.213	0.286

OPS5 (Weaver)

Algorithm (Cache Size)	Total miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (512k)	0.146	0.145	0.165	0.170	0.169
mark-and-sweep (512k)	0.147	0.147	0.200	0.194	0.152
stop-and-copy (1M)	0.081	0.078	0.083	0.082	0.082
mark-and-sweep (1M)	0.077	0.079	0.130	0.126	0.087
stop-and-copy (2M)	0.051	0.047	0.056	0.056	0.056
mark-and-sweep (2M)	0.041	0.044	0.095	0.091	0.052

PMA

Algorithm (Cache Size)	Total miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (512k)	0.566	0.548	0.584	0.601	0.626
mark-and-sweep (512k)	0.377	0.406	0.476	0.522	0.550
stop-and-copy (1M)	0.498	0.475	0.456	0.502	0.531
mark-and-sweep (1M)	0.176	0.224	0.288	0.363	0.435
stop-and-copy (2M)	0.262	0.204	0.178	0.278	0.445
mark-and-sweep (2M)	0.135	0.174	0.209	0.266	0.315

Table C.33: Total Cache Miss Rates for Three Collection Algorithms.

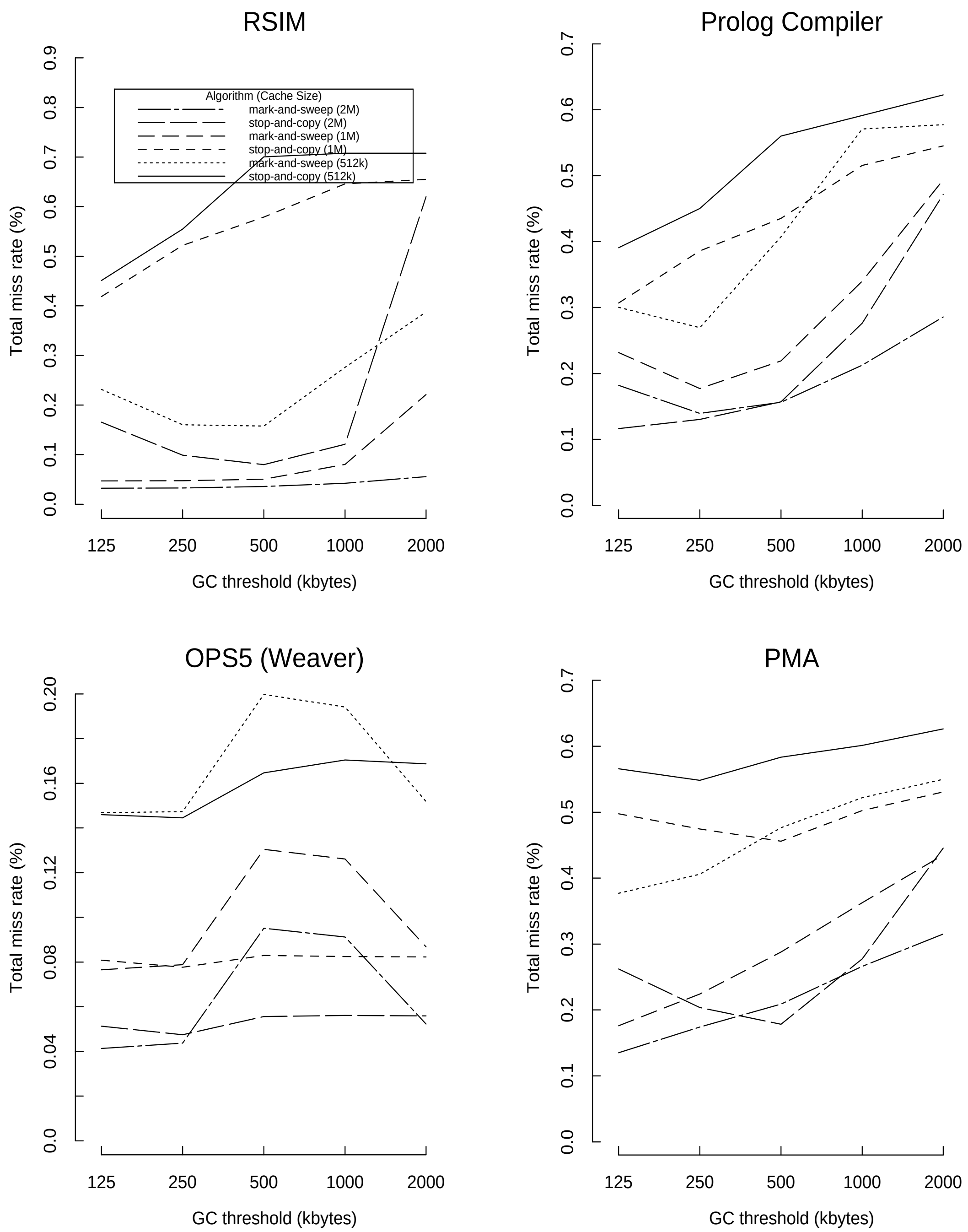

Figure C.28: Total Cache Miss Rates for Three Collection Algorithms.

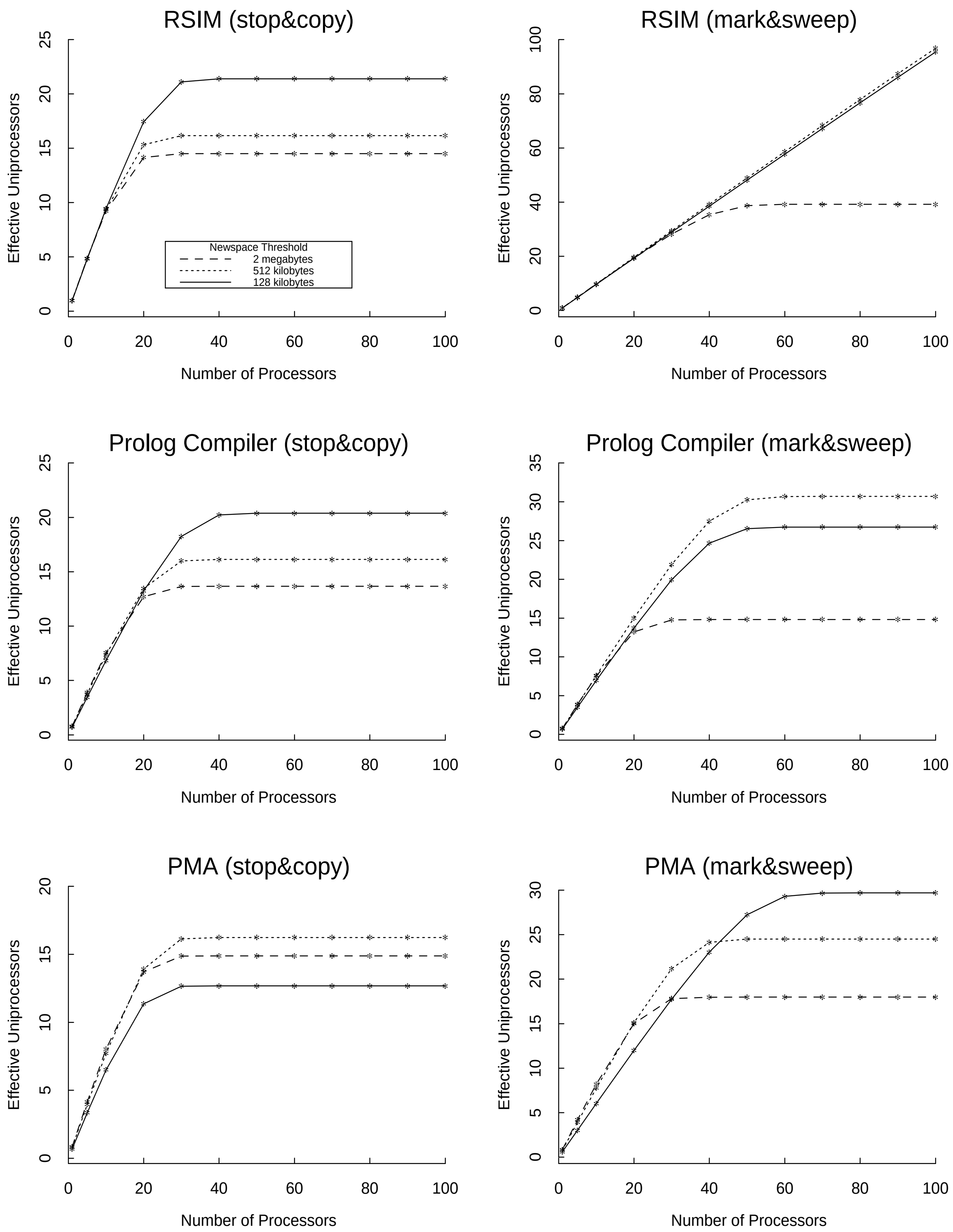

Figure C.29: Maximum Effective Uniprocessors for Different Algorithms and Threshold Sizes. Garbage collection induced overhead is taken into account.

RSIM (stop&copy)

Actual Processors	Effective Uniprocessors		
	$th = 125$	$th = 500$	$th = 2000$
1	0.97	0.99	0.99
5	4.81	4.86	4.86
10	9.45	9.36	9.25
20	17.44	15.33	14.16
30	21.11	16.16	14.51
40	21.40	16.17	14.51
50	21.40	16.17	14.51
60	21.40	16.17	14.51
70	21.40	16.17	14.51
80	21.40	16.17	14.51
90	21.40	16.17	14.51
100	21.40	16.17	14.51

RSIM (mark&sweep)

Effective Uniprocessors		
$th = 125$	$th = 500$	$th = 2000$
0.97	0.98	0.99
4.83	4.91	4.92
9.65	9.82	9.80
19.29	19.62	19.35
28.92	29.42	28.24
38.54	39.19	35.34
48.13	48.94	38.73
57.70	58.66	39.23
67.23	68.33	39.25
76.71	77.94	39.25
86.13	87.46	39.25
95.45	96.84	39.25

Prolog Compiler (stop&copy)

Actual Processors	Effective Uniprocessors		
	$th = 125$	$th = 500$	$th = 2000$
1	0.69	0.76	0.79
5	3.45	3.76	3.89
10	6.85	7.38	7.54
20	13.25	13.48	12.71
30	18.26	15.99	13.65
40	20.23	16.14	13.66
50	20.39	16.14	13.66
60	20.39	16.14	13.66
70	20.39	16.14	13.66
80	20.39	16.14	13.66
90	20.39	16.14	13.66
100	20.39	16.14	13.66

Prolog Compiler (mark&sweep)

Effective Uniprocessors		
$th = 125$	$th = 500$	$th = 2000$
0.70	0.77	0.78
3.50	3.82	3.86
6.97	7.61	7.53
13.73	15.02	13.22
19.95	21.93	14.78
24.66	27.51	14.81
26.55	30.25	14.81
26.74	30.68	14.81
26.75	30.69	14.81
26.75	30.69	14.81
26.75	30.69	14.81
26.75	30.69	14.81

PMA (stop&copy)

Actual Processors	Effective Uniprocessors		
	$th = 125$	$th = 500$	$th = 2000$
1	0.68	0.80	0.84
5	3.33	3.94	4.14
10	6.49	7.72	8.04
20	11.36	13.92	13.71
30	12.66	16.14	14.87
40	12.68	16.24	14.88
50	12.68	16.24	14.88
60	12.68	16.24	14.88
70	12.68	16.24	14.88
80	12.68	16.24	14.88
90	12.68	16.24	14.88
100	12.68	16.24	14.88

PMA (mark&sweep)

Effective Uniprocessors		
$th = 125$	$th = 500$	$th = 2000$
0.61	0.79	0.85
3.03	3.91	4.19
6.04	7.77	8.21
11.99	15.12	15.01
17.74	21.19	17.81
23.05	24.13	17.98
27.24	24.50	17.98
29.31	24.50	17.98
29.68	24.50	17.98
29.70	24.50	17.98
29.70	24.50	17.98
29.70	24.50	17.98

Table C.34: Maximum Effective Uniprocessors for Different Algorithms and Threshold Sizes. Garbage collection induced overhead is taken into account.

Appendix D

Tables

This appendix contains the results that are presented in the thesis as figures in a tabular form.

Lisp Application	Fraction of Bytes Allocated (%)				
	cons	symbol	vector	number	other
Lisp Compiler	53.66	0.75	28.40	14.67	2.52
Curare	53.14	0.05	20.12	0.71	25.97
Boyer Moore TP	92.43	0.06	4.82	1.96	0.73
RL	62.50	0.01	35.72	0.04	1.72

Table D.1: Object Allocations for Test Programs (by type and size).

Lisp Application	Fraction of Objects Allocated (%)				
	cons	symbol	vector	number	other
Lisp Compiler	78.79	0.37	4.63	15.19	1.01
Curare	78.15	0.03	10.25	0.65	10.93
Boyer Moore TP	98.16	0.02	0.34	1.24	0.25
RL	81.99	0.01	17.64	0.05	0.32

Table D.2: Object Allocations for Test Programs (by type and number).

Lisp Application	Fraction of References (%)				
	cons	symbol	vector	number	other
Lisp Compiler	63.98	12.73	17.73	2.62	2.20
Curare	86.46	1.89	4.90	0.09	2.25
Boyer Moore TP	71.45	24.51	3.29	0.13	0.23
RL	57.32	10.60	29.13	0.01	0.83

Table D.3: Object References by Type for Test Programs.

Lisp Application	Fraction of References (%)				
	loadp	load	storep	store	storei
Lisp Compiler	74.80	0.59	5.57	1.16	17.88
Curare	90.80	0.31	1.55	0.05	7.28
Boyer Moore TP	88.09	0.14	7.72	0.06	3.99
RL	71.53	0.43	9.04	0.07	18.93

Table D.4: Object References by Instruction Type for Test Programs.

Lisp Compiler

Time	Allocation Rate (kbytes/sec)			
(sec)	cons	vector	number	other
5.6	188.8	115.8	44.4	10.4
11.2	198.7	133.3	36.6	14.4
16.7	209.8	140.6	35.5	13.6
22.3	140.1	64.7	1.3	7.9
27.9	198.0	106.8	54.1	14.8
33.5	210.7	120.2	26.5	12.6
39.1	205.1	131.8	34.3	12.7
44.7	216.0	133.5	27.9	11.2
50.2	218.1	138.4	28.8	11.7
55.8	120.8	72.3	168.7	7.9
61.4	146.0	95.3	84.6	7.4
67.0	186.9	100.6	40.1	15.1
72.6	184.3	89.9	35.5	12.4
78.2	178.9	91.7	30.3	11.4
83.7	139.5	73.5	130.2	8.9
89.3	204.0	123.5	36.7	11.7
94.9	206.7	112.4	40.2	13.5
100.5	204.0	105.4	41.9	15.6
106.1	201.4	101.3	36.1	13.9
111.6	221.7	84.3	19.6	9.1
117.2	148.5	59.6	0.0	7.1
122.8	186.6	98.9	55.4	16.1
128.4	205.0	109.5	30.6	11.8
134.0	197.8	74.1	15.1	9.4
139.6	205.1	123.5	42.5	12.2
145.1	211.6	65.6	25.1	11.6
150.7	219.0	54.2	0.2	8.2
156.3	230.1	129.8	54.3	15.7
161.9	233.9	124.3	38.5	13.8
167.5	4.5	0.1	321.3	0.5

Curare

Time	Allocation Rate (kbytes/sec)			
(sec)	cons	vector	number	other
3.9	208.5	98.7	15.6	137.0
7.7	106.2	62.7	5.7	133.4
11.6	87.8	82.2	1.2	219.5
15.4	288.6	135.9	0.7	119.6
19.3	256.3	77.3	0.1	78.6
23.2	316.0	68.2	0.0	87.2
27.0	176.3	45.0	0.2	43.1
30.9	146.2	25.3	0.2	22.5
34.8	69.8	22.9	0.0	11.3
38.6	34.6	9.0	0.1	7.5
42.5	36.3	11.3	0.0	6.2
46.3	15.4	2.8	0.0	1.8
50.2	15.2	0.2	0.0	1.2
54.1	26.6	9.4	0.0	5.3
57.9	20.3	5.1	0.0	2.1
61.8	21.6	2.1	0.3	4.6
65.7	29.3	11.7	0.0	2.2
69.5	12.3	1.2	0.0	2.7
73.4	21.3	5.7	0.0	0.7
77.2	24.5	5.6	0.0	2.2
81.1	24.0	5.8	0.0	3.3
85.0	24.6	5.0	0.0	3.0
88.8	20.2	6.3	0.0	1.0
92.7	23.9	2.1	0.0	1.8
96.6	19.7	5.4	0.0	0.9
100.4	21.7	5.6	0.0	0.9
104.3	14.1	4.9	0.0	0.5
108.1	11.2	0.1	0.0	0.0
112.0	97.2	69.4	5.3	87.0
115.9	97.6	71.4	1.0	123.0

Table D.5: Object Allocation Rates as a Function of Time.

Boyer Moore TP

Time (sec)	Allocation Rate (kbytes/sec)			
	cons	vector	number	other
4.6	147.9	72.7	1.1	6.6
9.2	114.2	8.7	5.1	2.9
13.9	69.8	3.0	1.8	0.2
18.5	78.8	2.6	3.0	0.6
23.1	48.4	0.0	0.2	0.0
27.7	66.1	0.1	0.9	0.0
32.3	91.3	2.9	4.9	0.8
36.9	45.3	0.6	0.3	0.2
41.6	67.1	2.6	2.0	0.7
46.2	71.5	0.1	1.4	0.0
50.8	64.6	0.1	1.0	0.0
55.4	90.7	2.5	1.5	0.6
60.0	79.6	0.0	0.1	0.0
64.6	94.9	0.0	0.1	0.0
69.3	97.5	4.9	2.8	0.6
73.9	47.9	0.0	0.1	0.1
78.5	41.3	0.0	0.0	0.0
83.1	54.4	0.1	1.0	0.2
87.7	86.6	2.7	4.2	1.1
92.3	73.8	0.1	1.3	0.2
97.0	77.4	0.1	1.6	0.3
101.6	74.8	2.5	2.2	0.3
106.2	49.8	0.1	0.5	0.1
110.8	68.9	0.2	2.8	0.3
115.4	40.6	0.0	0.2	0.0
120.1	53.2	2.4	0.4	0.0
124.7	69.1	0.1	0.9	0.2
129.3	71.8	0.1	0.8	0.2
133.9	64.3	0.1	0.5	0.2
138.5	60.3	3.4	3.1	1.9

RL

Time (sec)	Allocation Rate (kbytes/sec)			
	cons	vector	number	other
7.2	141.9	83.0	1.0	17.5
14.4	237.0	180.3	0.0	1.0
21.6	210.9	124.7	0.0	0.6
28.8	253.5	146.1	0.0	1.3
36.0	246.5	144.7	0.0	1.2
43.2	202.1	110.4	0.0	16.2
50.4	174.6	65.2	1.4	26.9
57.6	99.2	41.3	1.9	12.5
64.8	75.7	51.7	0.0	1.0
72.0	279.9	154.6	0.0	0.7
79.2	263.6	147.1	0.0	0.4
86.5	279.9	162.0	0.0	1.4
93.7	260.7	140.3	0.0	0.5
100.9	227.6	126.5	0.0	0.4
108.1	283.2	167.6	0.0	0.6
115.3	208.9	136.2	0.0	1.4
122.5	207.6	137.6	0.0	1.6
129.7	190.9	140.5	0.0	1.4
136.9	264.2	158.9	0.0	1.3
144.1	252.3	148.3	0.0	1.2
151.3	259.9	153.7	0.0	0.2
158.5	238.0	139.3	0.0	0.7
165.7	322.3	175.5	0.0	0.6
172.9	286.3	158.6	0.0	0.4
180.1	267.9	145.0	0.0	1.1
187.3	304.8	170.8	0.0	0.8
194.5	278.6	149.8	0.0	0.5
201.7	244.3	131.8	0.0	0.5
208.9	261.2	129.6	0.0	0.5
216.1	109.0	40.8	0.1	98.4

Table D.6: Object Allocation Rates as a Function of Time.

Lisp Compiler

Time (sec)	Net Rate (kbytes/sec)			
	cons	vector	number	other
5.6	7.7	1.5	0.0	8.9
11.2	7.9	1.3	0.0	10.9
16.7	3.9	0.9	0.0	9.9
22.3	62.4	21.4	0.0	4.5
27.9	-23.5	-20.6	0.0	10.9
33.5	4.2	1.0	0.0	8.3
39.1	-0.4	0.1	0.0	8.4
44.7	3.9	0.5	0.0	5.3
50.2	3.4	0.6	0.0	6.4
55.8	-8.0	0.1	0.0	-10.8
61.4	10.1	7.8	0.0	1.2
67.0	-8.6	1.9	0.0	7.5
72.6	-28.9	-8.2	0.0	6.7
78.2	-2.2	0.6	0.0	-3.1
83.7	-4.0	-0.2	-0.0	-22.3
89.3	10.5	0.9	0.0	2.2
94.9	2.5	1.2	0.0	1.2
100.5	2.1	-0.2	0.0	4.8
106.1	-7.2	-3.8	-0.0	1.2
111.6	30.4	2.0	0.0	5.6
117.2	58.6	29.0	0.0	2.0
122.8	-80.8	-28.2	0.0	4.9
128.4	-2.8	-1.6	0.0	4.4
134.0	18.0	0.3	0.0	2.7
139.6	-4.7	14.0	0.0	8.6
145.1	10.8	-15.6	-0.0	3.9
150.7	76.6	32.7	0.0	0.3
156.3	-20.9	-3.3	0.0	2.6
161.9	-43.0	-0.3	-0.0	2.1
167.5	-78.1	-35.9	-0.0	-99.4

Curare

Time (sec)	Net Rate (kbytes/sec)			
	cons	vector	number	other
3.9	15.3	10.3	0.0	7.9
7.7	7.1	21.7	0.0	26.2
11.6	0.8	6.1	0.0	10.9
15.4	35.5	34.9	0.0	0.5
19.3	17.0	9.0	0.0	0.0
23.2	16.7	8.9	0.0	0.0
27.0	11.6	6.4	0.0	-0.0
30.9	6.9	3.8	0.0	0.0
34.8	12.8	7.8	0.0	0.0
38.6	5.4	2.7	0.0	0.0
42.5	5.5	3.5	0.0	0.0
46.3	1.8	0.9	0.0	0.0
50.2	-0.5	0.0	0.0	0.0
54.1	5.0	1.1	0.0	0.0
57.9	2.8	1.4	0.0	0.0
61.8	-1.9	0.0	0.0	0.0
65.7	9.6	4.1	0.0	0.0
69.5	-0.8	0.5	0.0	0.0
73.4	4.8	2.4	0.0	0.0
77.2	2.0	1.5	0.0	0.0
81.1	3.4	2.1	0.0	0.0
85.0	2.7	1.6	0.0	0.0
88.8	-32.4	-25.2	-0.0	-0.3
92.7	0.9	0.6	0.0	0.0
96.6	4.6	1.9	0.0	0.0
100.4	2.8	1.9	0.0	0.0
104.3	-0.5	1.7	0.0	0.0
108.1	0.2	0.0	0.0	0.0
112.0	3.1	13.8	0.0	12.6
115.9	-142.4	-125.4	-0.0	-57.8

Table D.7: Net Allocation Rates as a Function of Time.

Boyer Moore TP

Time	Net Rate (kbytes/sec)			
(sec)	cons	vector	number	other
4.6	21.5	2.9	0.0	2.7
9.2	10.0	0.4	0.0	1.0
13.9	2.6	0.1	0.0	0.1
18.5	-0.2	0.0	0.0	0.3
23.1	0.4	0.0	0.0	0.0
27.7	-0.5	-0.0	-0.0	0.0
32.3	1.5	-0.5	0.0	0.2
36.9	-0.1	0.6	0.0	0.0
41.6	2.2	0.0	0.0	0.1
46.2	1.1	0.0	0.0	0.0
50.8	-0.0	0.0	0.0	0.0
55.4	3.5	0.0	0.0	0.1
60.0	-1.2	-0.0	-0.0	0.0
64.6	-0.3	0.0	0.0	0.0
69.3	1.8	0.0	0.0	0.2
73.9	0.5	0.0	0.0	0.0
78.5	0.0	0.0	0.0	0.0
83.1	-0.2	0.0	0.0	0.0
87.7	3.2	0.1	0.0	0.2
92.3	0.1	0.0	0.0	0.0
97.0	0.5	0.0	0.0	0.0
101.6	1.5	0.0	0.0	0.0
106.2	-0.6	0.0	0.0	0.0
110.8	3.7	0.0	0.0	0.0
115.4	-0.5	0.0	0.0	0.0
120.1	-0.0	0.0	0.0	0.0
124.7	-0.0	0.0	0.0	0.0
129.3	-1.4	-0.0	-0.0	-0.1
133.9	-0.1	0.0	0.0	0.0
138.5	-48.7	-3.6	-0.2	-4.8

RL

Time	Net Rate (kbytes/sec)			
(sec)	cons	vector	number	other
7.2	27.8	12.1	0.0	1.5
14.4	19.8	21.7	0.0	0.0
21.6	12.0	14.1	0.0	0.0
28.8	-23.3	-29.4	0.0	0.0
36.0	8.5	8.3	0.0	0.0
43.2	-6.0	-6.4	0.0	0.1
50.4	2.5	-7.7	-0.0	0.2
57.6	37.1	3.7	0.0	1.0
64.8	18.4	19.8	0.0	0.0
72.0	14.5	14.5	0.0	-0.0
79.2	-13.6	-4.8	0.0	-0.0
86.5	10.4	0.1	0.0	-0.3
93.7	10.2	11.2	0.0	-0.4
100.9	9.6	9.9	0.0	0.0
108.1	-40.2	-42.7	0.0	0.0
115.3	-5.6	-8.0	0.0	0.0
122.5	3.0	6.4	0.0	0.0
129.7	0.7	-3.8	0.0	0.0
136.9	11.0	12.0	0.0	0.0
144.1	5.0	4.1	0.0	0.0
151.3	19.3	23.5	0.0	0.0
158.5	-23.1	-28.4	0.0	0.0
165.7	-47.3	19.2	-0.0	-0.7
172.9	12.5	9.9	-0.0	-1.0
180.1	-29.4	-33.9	0.0	-0.0
187.3	18.7	21.1	0.0	0.0
194.5	13.3	14.6	0.0	0.0
201.7	8.9	10.1	0.0	0.0
208.9	-45.1	-51.1	0.0	-0.0
216.1	-29.6	-19.8	0.0	-0.2

Table D.8: Net Allocation Rates as a Function of Time.

Lisp Compiler

Implementation Method	Write Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	0.1	0.0	0.0	0.0	0.0
write trap/fast OS	15.9	10.1	5.8	3.5	2.0
write trap/slow OS	18.3	11.6	6.7	4.0	2.3
software test	5.4	5.4	5.1	5.0	4.9

Curare

Implementation Method	Write Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	0.1	0.1	0.1	0.1	0.0
write trap/fast OS	6.4	3.4	2.1	1.3	0.9
write trap/slow OS	7.4	3.9	2.4	1.5	1.0
software test	1.7	1.6	1.5	1.5	1.3

Boyer Moore TP

Implementation Method	Write Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	0.3	0.2	0.0	0.0	0.0
write trap/fast OS	5.1	2.9	1.6	1.0	0.6
write trap/slow OS	5.9	3.3	1.9	1.1	0.7
software test	13.9	13.9	13.8	13.8	13.8

RL

Implementation Method	Write Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	3.1	2.8	2.6	1.8	0.8
write trap/fast OS	54.5	23.8	15.3	6.2	2.6
write trap/slow OS	62.7	27.4	17.6	7.1	3.0
software test	14.0	13.5	13.2	11.2	8.6

Table D.9: CPU Overhead for Write Barrier Implementations. Stop-and-copy is the garbage collection algorithm used in all cases.

Lisp Compiler

Implementation Method	Read Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	10.5	10.0	9.8	9.6	9.5
software test	42.7	42.7	42.6	42.3	42.2
modified eq/fast OS	60.6	39.0	29.8	24.3	20.5
modified eq/slow OS	235.0	130.5	85.6	57.9	39.4
scan stack/fast OS	512.8	272.3	169.0	104.8	62.3
scan stack/slow OS	687.1	363.8	224.8	138.4	81.2

Curare

Implementation Method	Read Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	11.5	11.4	11.4	11.4	11.4
software test	51.1	51.1	51.2	51.2	51.1
modified eq/fast OS	14.9	13.9	13.0	12.8	11.3
modified eq/slow OS	31.4	26.4	22.2	21.2	13.5
scan stack/fast OS	50.4	38.8	29.2	26.8	9.2
scan stack/slow OS	66.9	51.3	38.4	35.2	11.5

Boyer Moore TP

Implementation Method	Read Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	11.0	11.0	11.0	11.0	11.0
software test	49.5	49.6	49.6	49.6	49.6
modified eq/fast OS	14.1	13.3	13.0	12.8	12.6
modified eq/slow OS	20.5	16.7	15.0	14.1	13.3
scan stack/fast OS	22.9	14.1	10.1	8.2	6.4
scan stack/slow OS	29.3	17.4	12.1	9.5	7.1

RL

Implementation Method	Read Barrier CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
hardware trap	9.8	9.6	9.4	9.3	9.2
software test	41.3	41.2	40.8	40.7	40.6
modified eq/fast OS	42.6	37.6	32.1	29.5	27.4
modified eq/slow OS	138.4	114.0	86.5	73.7	63.2
scan stack/fast OS	287.4	231.2	167.5	138.1	113.8
scan stack/slow OS	383.2	307.6	221.9	182.3	149.6

Table D.10: CPU Overhead for Read Barrier Implementations. The results indicate the overhead of the read barrier for an incremental copying algorithm.

Lisp Compiler (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	5.1	4.7	4.6	4.6	4.5
scan	27.1	17.6	12.8	10.2	9.4
forward	33.1	21.4	15.4	12.1	11.1
transport	47.2	30.4	21.3	16.3	14.9
update	51.3	32.9	22.9	17.5	15.9

Curare (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	1.9	1.8	1.8	1.8	1.7
scan	7.0	5.5	5.0	4.4	3.8
forward	9.0	6.8	6.2	5.5	4.7
transport	13.3	9.9	8.8	7.7	6.5
update	14.4	10.7	9.5	8.3	7.0

Boyer Moore TP (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	1.4	1.4	1.4	1.3	1.2
scan	3.1	2.6	2.2	1.9	1.6
forward	3.6	2.9	2.5	2.1	1.8
transport	4.7	3.7	3.1	2.5	2.1
update	5.0	4.0	3.3	2.7	2.2

RL (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	4.8	5.0	5.3	5.3	5.3
scan	38.8	28.1	21.1	15.7	12.2
forward	45.0	33.0	24.7	18.3	14.3
transport	62.0	45.0	33.1	24.4	18.7
update	66.0	48.1	35.4	26.0	19.9

Table D.11: Cumulative CPU Overhead of Copying Collection. The algorithm used is stop-and-copy collection. Results for incremental copying are similar.

Lisp Compiler (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	6.1	6.0	5.9	5.8	5.6
sweep	12.1	11.1	10.4	10.2	9.8
stack	20.9	16.1	13.9	12.4	11.7
type	35.2	24.3	19.5	15.8	14.8
marking	48.4	31.9	24.4	18.7	17.6
relocate	50.9	33.4	25.5	19.7	18.4

Curare (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	2.7	2.6	2.6	2.4	2.1
sweep	5.2	4.9	4.7	4.3	3.6
stack	7.6	6.7	6.3	5.5	4.4
type	11.0	9.4	8.7	7.5	5.7
marking	14.1	11.8	10.7	9.4	7.0
relocate	15.1	12.6	11.4	10.0	7.5

Boyer Moore TP (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	1.6	1.6	1.5	1.4	1.0
sweep	3.0	2.8	2.6	2.4	1.7
stack	3.8	3.3	3.0	2.6	1.9
type	5.0	4.1	3.6	3.0	2.2
marking	6.2	4.9	4.2	3.4	2.5
relocate	6.4	5.0	4.3	3.5	2.5

RL (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
alloc	5.4	5.5	6.1	6.1	5.9
sweep	10.5	10.3	10.8	10.5	10.0
stack	22.1	18.8	16.7	14.3	12.8
type	39.6	31.6	25.7	20.2	17.2
marking	54.5	43.0	33.6	25.5	21.0
relocate	58.0	45.5	35.3	26.6	21.9

Table D.12: Cumulative CPU Overhead of Mark-and-Sweep Collection.

Lisp Compiler (incremental)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	50.2	32.6	23.0	17.7	12.7
write barrier	71.3	46.3	30.8	23.1	16.0
read barrier	306.2	176.8	116.4	81.0	55.4

RL (incremental)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	61.2	47.8	35.1	25.8	20.0
write barrier	104.1	71.1	49.5	32.4	22.8
read barrier	242.4	185.1	136.0	106.2	86.0

Lisp Compiler (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	51.3	32.9	22.9	17.5	15.9
write barrier	69.6	44.5	29.6	21.5	18.2

RL (stop&copy)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	66.0	48.1	35.3	26.0	19.9
write barrier	128.7	75.4	52.9	33.2	22.9

Lisp Compiler (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	50.9	33.4	25.5	19.7	18.4
write barrier	61.5	39.8	29.3	22.0	19.9
indirect vectors	63.4	41.5	30.9	23.6	21.5

RL (mark&sweep)

Overhead Source	Cumulative CPU Overhead (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
base overhead	58.0	45.5	35.3	26.6	21.9
write barrier	108.5	74.5	54.4	33.7	24.6
indirect vectors	111.0	76.9	56.7	36.1	26.9

Table D.13: Cumulative CPU Overhead for Three Algorithms.

Age (sec)	**Lisp Compiler** Fraction of references (%)					**Curare** Fraction of references (%)				
	cons	vector	number	other	total	cons	vector	number	other	total
0.000004	19.4	3.3	70.5	7.7	17.1	5.1	11.3	37.7	24.7	6.0
0.000008	7.2	1.7	14.7	3.8	6.1	1.4	3.4	12.3	9.3	1.7
0.000016	0.8	3.2	14.7	7.5	2.1	0.3	4.9	11.9	13.9	0.9
0.000032	1.4	5.6	0.0	7.8	2.5	0.2	4.9	0.0	18.1	0.9
0.000064	1.1	6.0	0.0	5.6	2.4	0.3	5.1	0.0	19.2	1.0
0.000128	1.3	3.4	0.0	4.5	1.9	0.3	3.6	0.0	1.0	0.5
0.000256	1.1	0.7	0.0	3.6	1.1	0.3	3.8	1.2	0.3	0.5
0.000512	1.3	0.8	0.0	2.7	1.2	0.3	1.3	2.0	0.3	0.4
0.001024	1.4	1.4	0.0	4.0	1.4	0.3	2.7	4.3	0.3	0.5
0.002048	1.6	2.3	0.0	2.6	1.8	0.4	4.1	6.7	0.2	0.6
0.004096	2.1	1.6	0.0	1.0	1.9	0.8	3.2	9.7	0.3	0.9
0.008192	2.5	2.6	0.0	1.5	2.4	1.1	3.7	11.5	0.4	1.3
0.016384	4.0	3.9	0.0	1.9	3.8	1.1	3.6	2.6	0.5	1.2
0.032768	5.9	4.1	0.0	1.8	5.2	1.4	4.6	0.0	1.0	1.6
0.065536	7.0	4.1	0.0	1.4	5.8	2.0	5.1	0.0	1.5	2.2
0.131072	6.5	5.6	0.0	1.0	5.9	3.1	4.6	0.0	1.5	3.2
0.262144	6.1	5.6	0.0	1.1	5.6	4.5	4.8	0.0	0.1	4.4
0.524288	6.7	2.9	0.0	1.7	5.4	6.5	4.9	0.0	0.0	6.2
1.048580	6.5	3.9	0.0	2.6	5.5	8.9	2.4	0.0	0.0	8.3
2.097150	6.1	6.9	0.0	4.6	6.0	9.6	2.6	0.0	0.1	9.0
4.194300	3.1	6.8	0.0	7.5	4.0	10.6	4.3	0.0	0.0	10.0
8.388610	1.5	8.5	0.0	10.4	3.4	11.2	3.1	0.0	0.1	10.4
16.777200	0.9	4.0	0.0	4.4	1.7	8.5	2.3	0.0	0.3	8.0
33.554400	1.4	3.1	0.0	1.7	1.8	8.2	2.3	0.0	0.0	7.7
67.108900	1.5	3.6	0.0	2.8	2.0	6.1	1.0	0.0	0.0	5.7
134.218000	1.4	4.3	0.0	3.7	2.1	5.9	1.0	0.0	0.0	5.5
268.435000	0.1	0.1	0.0	1.2	0.1	1.3	1.2	0.0	7.0	1.5
536.871000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table D.14: Age Distribution of Objects Referenced by Object Type.

Age (sec)	Boyer Moore TP: Fraction of references (%)					RL: Fraction of references (%)				
	cons	vector	number	other	total	cons	vector	number	other	total
0.000004	5.4	0.4	36.4	4.8	5.3	22.6	9.8	68.4	8.7	18.3
0.000008	2.2	0.2	12.4	2.4	2.2	4.5	4.8	5.1	3.8	4.6
0.000016	0.8	0.4	19.5	4.1	0.8	0.4	5.4	9.1	6.3	2.1
0.000032	0.6	0.7	19.6	5.1	0.6	1.5	2.0	8.3	8.4	1.7
0.000064	0.6	1.3	0.0	6.1	0.6	0.2	2.0	0.0	9.7	0.9
0.000128	0.7	0.8	3.4	6.0	0.8	0.6	0.6	0.0	9.3	0.7
0.000256	1.0	0.1	0.0	4.4	1.0	0.9	0.7	0.0	13.6	0.9
0.000512	2.5	0.1	0.0	3.2	2.4	0.9	0.3	0.0	18.6	0.8
0.001024	2.5	0.2	0.0	3.7	2.4	0.7	0.5	0.0	0.7	0.6
0.002048	1.8	0.3	0.0	3.4	1.7	0.4	0.5	0.0	0.0	0.4
0.004096	2.8	0.6	0.0	2.3	2.7	0.4	0.7	0.0	3.5	0.5
0.008192	4.0	1.0	0.0	1.0	3.9	0.5	0.9	0.1	9.4	0.7
0.016384	4.2	1.0	0.0	0.5	4.0	0.9	1.0	0.1	1.9	0.9
0.032768	4.0	1.1	0.0	0.3	3.8	1.4	1.6	0.2	1.8	1.4
0.065536	4.1	2.0	0.0	0.2	4.0	2.1	1.5	0.5	1.5	1.9
0.131072	4.1	2.4	0.0	0.2	4.0	3.0	1.4	0.3	0.1	2.4
0.262144	4.7	1.4	0.0	0.5	4.5	3.8	1.5	1.3	0.1	3.0
0.524288	4.0	2.7	0.0	0.6	4.0	4.4	2.9	2.4	0.1	3.9
1.048580	3.8	3.8	0.1	0.4	3.8	5.3	4.8	0.8	0.3	5.1
2.097150	3.8	7.5	0.3	0.8	3.9	6.5	6.0	1.7	0.3	6.3
4.194300	3.3	4.9	0.2	0.1	3.4	6.8	7.0	1.6	0.0	6.8
8.388610	3.3	0.7	0.9	1.3	3.2	9.3	11.4	0.1	0.0	9.9
16.777200	3.2	2.3	0.2	3.0	3.2	9.9	14.1	0.0	0.2	11.2
33.554400	3.8	8.9	0.5	2.8	4.0	8.5	13.0	0.0	0.0	10.0
67.108900	5.4	10.2	0.9	4.5	5.6	3.1	4.5	0.0	0.1	3.5
134.218000	7.9	16.0	1.8	9.5	8.2	0.5	0.6	0.0	0.5	0.6
268.435000	15.3	28.6	3.7	20.8	15.9	0.7	0.5	0.0	0.3	0.6
536.871000	0.1	0.4	0.0	8.0	0.2	0.2	0.1	0.0	0.7	0.1

Table D.15: Age Distribution of Objects Referenced by Object Type.

Lisp Compiler (incremental)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	20.9	19.5	66.0	99.1	118.3
4 megabytes	9.0	11.6	25.6	23.2	91.2
5 megabytes	2.4	8.7	12.1	2.4	70.0
6 megabytes	0.4	6.1	6.1	0.1	32.7

RL (incremental)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	147.4	10.9	44.6	107.8	100.7
4 megabytes	38.4	5.7	15.2	37.6	70.2
5 megabytes	11.7	1.8	6.1	14.3	52.0
6 megabytes	3.4	0.5	1.4	6.5	23.8

Lisp Compiler (stop&copy)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	9.9	19.6	59.9	103.0	124.7
4 megabytes	2.3	10.0	18.1	24.1	92.9
5 megabytes	0.3	3.1	9.4	1.8	71.8
6 megabytes	0.0	0.8	4.1	0.1	36.0

RL (stop&copy)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	21.7	8.8	57.1	104.2	90.3
4 megabytes	12.9	3.0	10.7	38.4	67.7
5 megabytes	5.3	1.2	3.5	12.2	49.9
6 megabytes	0.1	0.0	0.2	4.2	22.9

Lisp Compiler (mark&sweep)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	13.8	10.4	7.6	25.0	91.9
4 megabytes	5.9	4.5	3.6	8.3	44.3
5 megabytes	0.3	0.4	0.4	4.1	25.7
6 megabytes	0.0	0.1	0.0	1.6	12.2

RL (mark&sweep)

Memory Size	Page faults per second				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
3 megabytes	5.2	10.7	9.7	23.1	58.5
4 megabytes	1.0	4.7	1.2	10.8	22.8
5 megabytes	0.0	1.4	0.0	4.6	6.1
6 megabytes	0.0	0.0	0.0	1.3	1.4

Table D.16: Page Fault Rates for Different Collection Algorithms.

Lisp Compiler (incremental)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	1642	1891	1966	1579	1874
5 faults/sec	1267	1638	1712	1278	1651
10 faults/sec	1149	1579	1606	1233	1636
20 faults/sec	993	1157	1356	1141	1599

RL (incremental)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	1906	1685	1807	1966	1849
5 faults/sec	1764	1244	1421	1649	1797
10 faults/sec	1409	1101	1349	1557	1756
20 faults/sec	1293	792	1111	1417	1692

Lisp Compiler (stop&copy)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	1516	1719	1897	1586	2636
5 faults/sec	1016	1316	1641	1273	2019
10 faults/sec	959	1190	1507	1217	1982
20 faults/sec	757	1026	1254	1113	1698

RL (stop&copy)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	1765	1516	1666	2095	1833
5 faults/sec	1348	1095	1304	1665	1707
10 faults/sec	1286	895	1229	1530	1679
20 faults/sec	1097	743	1055	1413	1627

Lisp Compiler (mark&sweep)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	1354	1445	1352	1806	2184
5 faults/sec	1135	1086	1126	1503	1855
10 faults/sec	1046	1010	917	1183	1681
20 faults/sec	904	781	694	983	1571

RL (mark&sweep)

Tolerance Level	Memory Needed (4K pages)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
0 faults/sec	1246	1433	1277	1656	1689
5 faults/sec	894	1184	962	1395	1506
10 faults/sec	781	1011	880	1252	1380
20 faults/sec	673	791	754	1034	1193

Table D.17: Memory Sizes Required for Different Collection Algorithms.

Lisp Compiler (stop&copy)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	5.72	5.78	5.31	5.58	5.33
128 kilobytes	4.81	5.00	4.51	4.69	4.48
256 kilobytes	3.22	4.19	3.75	3.88	3.75
512 kilobytes	2.54	3.12	3.20	3.31	3.24
1 megabyte	2.28	2.71	2.57	2.94	2.93
2 megabytes	0.77	0.92	0.68	1.12	2.68

Curare (stop&copy)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	3.82	3.75	4.73	5.33	5.82
128 kilobytes	2.09	2.11	2.80	2.92	3.27
256 kilobytes	1.13	1.09	1.63	1.75	1.55
512 kilobytes	0.75	0.68	1.05	0.82	0.81
1 megabyte	0.33	0.65	0.73	0.47	0.39
2 megabytes	0.14	0.35	0.42	0.34	0.24

Boyer Moore TP (stop&copy)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	1.44	1.43	1.54	1.60	1.43
128 kilobytes	0.97	1.04	0.99	1.20	0.92
256 kilobytes	0.63	0.83	0.73	0.80	0.73
512 kilobytes	0.55	0.54	0.63	0.70	0.63
1 megabyte	0.45	0.50	0.52	0.57	0.58
2 megabytes	0.21	0.20	0.26	0.45	0.53

RL (stop&copy)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	7.45	6.43	6.70	6.78	7.17
128 kilobytes	6.01	5.34	5.73	5.67	6.01
256 kilobytes	4.32	4.28	4.83	4.68	4.97
512 kilobytes	3.33	3.14	3.99	3.80	4.03
1 megabyte	2.85	2.75	3.05	3.05	3.23
2 megabytes	1.00	1.12	1.52	1.78	2.57

Table D.18: Cache Miss Rates for Stop-and-Copy Collection.

Lisp Compiler (mark&sweep)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	4.90	4.80	4.87	5.00	5.11
128 kilobytes	3.76	3.93	4.09	4.28	4.40
256 kilobytes	2.31	2.82	3.44	3.67	3.82
512 kilobytes	0.97	1.23	2.28	3.11	3.31
1 megabyte	0.67	0.73	1.52	2.22	2.92
2 megabytes	0.63	0.68	1.24	1.68	2.23

Curare (mark&sweep)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	5.19	5.16	6.18	6.46	7.29
128 kilobytes	2.72	2.86	3.46	3.20	4.45
256 kilobytes	1.79	1.42	1.52	1.50	2.52
512 kilobytes	1.00	0.68	0.75	0.79	1.21
1 megabyte	0.88	0.60	0.24	0.33	0.53
2 megabytes	0.52	0.16	0.24	0.32	0.28

Boyer Moore TP (mark&sweep)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	1.41	1.39	1.47	1.32	1.37
128 kilobytes	0.82	1.03	1.08	0.89	1.06
256 kilobytes	0.51	0.54	0.73	0.71	0.81
512 kilobytes	0.21	0.29	0.43	0.59	0.66
1 megabyte	0.15	0.21	0.25	0.33	0.58
2 megabytes	0.14	0.20	0.24	0.27	0.49

RL (mark&sweep)

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
64 kilobytes	9.38	8.09	7.26	6.15	6.10
128 kilobytes	7.82	6.58	5.63	4.81	4.80
256 kilobytes	6.12	4.95	4.34	3.64	3.71
512 kilobytes	3.70	3.13	3.10	2.83	2.72
1 megabyte	2.85	2.02	1.77	1.68	2.06
2 megabytes	1.76	1.60	1.51	1.46	1.28

Table D.19: Cache Miss Rates for Mark-and-Sweep Collection.

Lisp Compiler

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (64k)	5.72	5.78	5.31	5.58	5.33
mark-and-sweep (64k)	4.90	4.80	4.87	5.00	5.11
incremental (64k)	6.05	5.52	5.33	5.28	5.42
stop-and-copy (256k)	3.22	4.19	3.75	3.88	3.75
mark-and-sweep (256k)	2.31	2.82	3.44	3.67	3.82
incremental (256k)	3.81	3.82	3.71	3.68	3.75
stop-and-copy (1M)	2.28	2.71	2.57	2.94	2.93
mark-and-sweep (1M)	0.67	0.73	1.52	2.22	2.92
incremental (1M)	3.05	2.79	2.72	2.89	2.96

Curare

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (64k)	3.82	3.75	4.73	5.33	5.82
mark-and-sweep (64k)	5.19	5.16	6.18	6.46	7.29
incremental (64k)	3.82	3.87	4.00	5.29	5.02
stop-and-copy (256k)	1.13	1.09	1.63	1.75	1.55
mark-and-sweep (256k)	1.79	1.42	1.52	1.50	2.52
incremental (256k)	1.46	1.45	1.61	1.41	1.52
stop-and-copy (1M)	0.33	0.65	0.73	0.47	0.39
mark-and-sweep (1M)	0.88	0.60	0.24	0.33	0.53
incremental (1M)	0.86	1.13	1.01	0.54	0.39

Boyer Moore TP

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (64k)	1.44	1.43	1.54	1.60	1.43
mark-and-sweep (64k)	1.41	1.39	1.47	1.32	1.37
incremental (64k)	1.52	1.51	1.38	1.36	1.35
stop-and-copy (256k)	0.63	0.83	0.73	0.80	0.73
mark-and-sweep (256k)	0.51	0.54	0.73	0.71	0.81
incremental (256k)	0.75	0.89	0.81	0.78	0.74
stop-and-copy (1M)	0.45	0.50	0.52	0.57	0.58
mark-and-sweep (1M)	0.15	0.21	0.25	0.33	0.58
incremental (1M)	0.58	0.58	0.59	0.58	0.59

RL

Cache Size	Cache miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (64k)	7.45	6.43	6.70	6.78	7.17
mark-and-sweep (64k)	9.38	8.09	7.26	6.15	6.10
incremental (64k)	7.26	7.02	6.91	7.07	6.97
stop-and-copy (256k)	4.32	4.28	4.83	4.68	4.97
mark-and-sweep (256k)	6.12	4.95	4.34	3.64	3.71
incremental (256k)	4.48	4.77	4.98	4.89	4.99
stop-and-copy (1M)	2.85	2.75	3.05	3.05	3.23
mark-and-sweep (1M)	2.85	2.02	1.77	1.68	2.06
incremental (1M)	3.80	3.41	3.35	3.27	3.40

Table D.20: Cache Miss Rates for Three Collection Algorithms.

Age (sec)	Lisp Compiler: Fraction surviving					Curare: Fraction surviving				
	cons	vector	number	other	total	cons	vector	number	other	total
0.000004	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.000008	0.84	1.00	0.42	1.00	0.78	0.58	0.60	0.65	0.86	0.61
0.000016	0.81	0.94	0.00	1.00	0.69	0.52	0.37	0.02	0.61	0.51
0.000032	0.77	0.60	0.00	0.99	0.65	0.47	0.29	0.02	0.61	0.47
0.000064	0.74	0.60	0.00	0.99	0.63	0.44	0.26	0.02	0.29	0.40
0.000128	0.71	0.43	0.00	0.97	0.59	0.40	0.21	0.02	0.25	0.36
0.000256	0.68	0.42	0.00	0.90	0.57	0.36	0.21	0.02	0.25	0.33
0.000512	0.66	0.42	0.00	0.73	0.55	0.34	0.21	0.02	0.25	0.31
0.001024	0.64	0.42	0.00	0.61	0.53	0.32	0.21	0.02	0.25	0.30
0.002048	0.63	0.42	0.00	0.52	0.52	0.31	0.20	0.02	0.25	0.29
0.004096	0.61	0.41	0.00	0.52	0.51	0.30	0.20	0.02	0.25	0.28
0.008192	0.60	0.39	0.00	0.52	0.49	0.28	0.19	0.01	0.25	0.27
0.016384	0.57	0.37	0.00	0.51	0.48	0.26	0.19	0.00	0.25	0.25
0.032768	0.53	0.32	0.00	0.51	0.44	0.24	0.19	0.00	0.25	0.24
0.065536	0.48	0.26	0.00	0.51	0.40	0.22	0.18	0.00	0.25	0.22
0.131072	0.41	0.18	0.00	0.51	0.33	0.21	0.18	0.00	0.25	0.21
0.262144	0.34	0.14	0.00	0.51	0.28	0.19	0.17	0.00	0.23	0.19
0.524288	0.29	0.10	0.00	0.51	0.24	0.17	0.16	0.00	0.20	0.17
1.048580	0.21	0.08	0.00	0.50	0.18	0.16	0.16	0.00	0.16	0.16
2.097150	0.10	0.06	0.00	0.50	0.09	0.14	0.15	0.00	0.15	0.14
4.194300	0.07	0.04	0.00	0.50	0.06	0.12	0.14	0.00	0.15	0.12
8.388610	0.05	0.04	0.00	0.50	0.05	0.10	0.14	0.00	0.09	0.10
16.777200	0.04	0.02	0.00	0.49	0.04	0.08	0.13	0.00	0.06	0.09
33.554400	0.03	0.02	0.00	0.45	0.03	0.08	0.12	0.00	0.06	0.08
67.108900	0.02	0.00	0.00	0.40	0.02	0.07	0.12	0.00	0.06	0.07
134.218000	0.00	0.00	0.00	0.01	0.00	0.06	0.10	0.00	0.06	0.06
268.435000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
536.871000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table D.21: Survival Distribution of Objects Referenced by Object Type.

Boyer Moore TP

Age (sec)	Fraction surviving cons	vector	number	other	total
0.000004	1.00	1.00	1.00	1.00	1.00
0.000008	0.86	0.99	0.69	1.00	0.86
0.000016	0.80	0.84	0.40	0.96	0.80
0.000032	0.77	0.83	0.11	0.79	0.77
0.000064	0.73	0.83	0.10	0.71	0.72
0.000128	0.68	0.06	0.01	0.50	0.67
0.000256	0.61	0.06	0.01	0.41	0.60
0.000512	0.53	0.06	0.01	0.37	0.52
0.001024	0.45	0.06	0.01	0.32	0.44
0.002048	0.39	0.06	0.01	0.31	0.38
0.004096	0.35	0.04	0.01	0.29	0.34
0.008192	0.30	0.04	0.01	0.28	0.30
0.016384	0.23	0.04	0.01	0.28	0.23
0.032768	0.17	0.04	0.01	0.28	0.17
0.065536	0.13	0.04	0.01	0.28	0.13
0.131072	0.11	0.04	0.01	0.28	0.11
0.262144	0.10	0.04	0.01	0.28	0.10
0.524288	0.09	0.04	0.01	0.28	0.08
1.048580	0.07	0.04	0.01	0.28	0.07
2.097150	0.06	0.03	0.01	0.28	0.06
4.194300	0.05	0.03	0.01	0.28	0.05
8.388610	0.04	0.03	0.00	0.28	0.04
16.777200	0.04	0.03	0.00	0.28	0.04
33.554400	0.03	0.03	0.00	0.28	0.03
67.108900	0.02	0.03	0.00	0.28	0.02
134.218000	0.02	0.02	0.00	0.26	0.02
268.435000	0.01	0.01	0.00	0.15	0.01
536.871000	0.00	0.00	0.00	0.00	0.00

RL

Fraction surviving cons	vector	number	other	total
1.00	1.00	1.00	1.00	1.00
0.45	0.99	0.16	0.88	0.55
0.42	0.11	0.13	0.85	0.36
0.30	0.11	0.02	0.84	0.27
0.29	0.11	0.02	0.58	0.26
0.28	0.09	0.02	0.43	0.25
0.27	0.09	0.02	0.21	0.24
0.27	0.09	0.02	0.21	0.23
0.26	0.09	0.02	0.20	0.23
0.26	0.08	0.02	0.20	0.23
0.26	0.08	0.02	0.18	0.23
0.25	0.08	0.02	0.14	0.22
0.25	0.08	0.02	0.12	0.22
0.24	0.08	0.02	0.10	0.21
0.23	0.08	0.02	0.07	0.20
0.22	0.08	0.02	0.07	0.19
0.21	0.08	0.02	0.07	0.18
0.18	0.07	0.02	0.07	0.16
0.15	0.07	0.02	0.07	0.13
0.11	0.06	0.02	0.07	0.10
0.09	0.05	0.02	0.07	0.08
0.06	0.04	0.02	0.07	0.06
0.04	0.03	0.02	0.06	0.04
0.03	0.01	0.02	0.06	0.03
0.02	0.00	0.02	0.06	0.01
0.01	0.00	0.01	0.05	0.01
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00

Table D.22: Survival Distribution of Objects Referenced by Object Type.

Lisp Compiler (stop&copy)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.075	0.121	0.173	0.243	0.350
copies = 1 (DIS)	0.101	0.155	0.222	0.325	0.487
copies = 3 (DIS)	0.131	0.198	0.297	0.451	0.699
copies = 10 (DIS)	0.172	0.278	0.446	0.718	1.092
copies = 3 (MARS)	0.146	0.201	0.256	0.373	0.648

Lisp Compiler (mark&sweep)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.075	0.121	0.173	0.243	0.350
copies = 1 (DIS)	0.092	0.142	0.202	0.290	0.426
copies = 3 (DIS)	0.115	0.172	0.249	0.366	0.551
copies = 10 (DIS)	0.155	0.237	0.361	0.555	0.838
copies = 3 (MARS)	0.129	0.171	0.248	0.345	0.506

Curare (stop&copy)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.051	0.095	0.169	0.304	0.536
copies = 1 (DIS)	0.077	0.143	0.255	0.457	0.803
copies = 3 (DIS)	0.112	0.211	0.379	0.678	1.200
copies = 10 (DIS)	0.171	0.327	0.601	1.099	1.938
copies = 3 (MARS)	0.117	0.173	0.303	0.485	0.698

Curare (mark&sweep)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.051	0.095	0.169	0.304	0.536
copies = 1 (DIS)	0.067	0.125	0.221	0.394	0.695
copies = 3 (DIS)	0.093	0.173	0.305	0.543	0.949
copies = 10 (DIS)	0.150	0.282	0.508	0.907	1.547
copies = 3 (MARS)	0.084	0.143	0.211	0.355	0.498

RL (stop&copy)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.046	0.082	0.137	0.221	0.353
copies = 1 (DIS)	0.072	0.122	0.202	0.327	0.507
copies = 3 (DIS)	0.103	0.175	0.294	0.467	0.703
copies = 10 (DIS)	0.154	0.274	0.463	0.718	1.026
copies = 3 (MARS)	0.164	0.237	0.345	0.512	0.726

RL (mark&sweep)

Copy Number	Pause Length (sec)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.046	0.082	0.137	0.221	0.353
copies = 1 (DIS)	0.061	0.105	0.174	0.279	0.437
copies = 3 (DIS)	0.084	0.141	0.233	0.368	0.559
copies = 10 (DIS)	0.131	0.224	0.367	0.560	0.805
copies = 3 (MARS)	0.114	0.202	0.273	0.396	0.622

Table D.23: Pause Lengths for Three Applications

Lisp Compiler (stop&copy)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.755	0.613	0.439	0.307	0.221
copies = 1 (DIS)	1.342	0.986	0.666	0.462	0.336
copies = 3 (DIS)	2.313	1.507	1.000	0.698	0.513
copies = 10 (DIS)	4.835	2.908	1.917	1.354	0.922
copies = 3 (MARS)	2.147	1.471	0.935	0.679	0.593

Lisp Compiler (mark&sweep)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.755	0.613	0.439	0.307	0.221
copies = 1 (DIS)	1.089	0.813	0.557	0.390	0.280
copies = 3 (DIS)	1.679	1.130	0.751	0.524	0.383
copies = 10 (DIS)	3.351	2.016	1.322	0.918	0.642
copies = 3 (MARS)	1.888	1.252	0.906	0.670	0.498

Curare (stop&copy)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.386	0.358	0.317	0.284	0.252
copies = 1 (DIS)	0.697	0.643	0.560	0.494	0.429
copies = 3 (DIS)	1.285	1.169	1.000	0.864	0.739
copies = 10 (DIS)	3.254	2.897	2.398	2.011	1.597
copies = 3 (MARS)	1.123	0.826	0.716	0.559	0.403

Curare (mark&sweep)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.386	0.358	0.317	0.284	0.252
copies = 1 (DIS)	0.557	0.517	0.450	0.398	0.350
copies = 3 (DIS)	0.914	0.827	0.710	0.613	0.525
copies = 10 (DIS)	2.252	1.972	1.640	1.359	1.065
copies = 3 (MARS)	0.813	0.683	0.501	0.408	0.287

RL (stop&copy)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.482	0.427	0.354	0.287	0.228
copies = 1 (DIS)	0.883	0.738	0.597	0.469	0.358
copies = 3 (DIS)	1.579	1.263	1.000	0.746	0.534
copies = 10 (DIS)	3.665	2.845	2.091	1.390	0.876
copies = 3 (MARS)	2.630	1.766	1.213	0.872	0.610

RL (mark&sweep)

Copy Number	Relative Overhead				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	0.482	0.427	0.354	0.287	0.228
copies = 1 (DIS)	0.700	0.594	0.484	0.382	0.297
copies = 3 (DIS)	1.107	0.897	0.712	0.544	0.398
copies = 10 (DIS)	2.457	1.878	1.387	0.949	0.627
copies = 3 (MARS)	1.758	1.477	0.960	0.675	0.522

Table D.24: Relative CPU Overhead for Three Applications

Lisp Compiler (stop&copy)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	28.864	23.497	16.857	11.807	8.496
copies = 1 (DIS)	20.484	12.323	7.074	5.221	4.015
copies = 3 (DIS)	14.367	7.290	5.167	3.839	3.011
copies = 10 (DIS)	7.957	5.215	3.642	2.718	1.289
copies = 3 (MARS)	9.166	7.481	4.213	3.367	3.243

Lisp Compiler (mark&sweep)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	28.864	23.497	16.857	11.807	8.496
copies = 1 (DIS)	24.710	17.872	12.139	8.666	6.307
copies = 3 (DIS)	20.166	13.255	9.050	6.374	4.819
copies = 10 (DIS)	14.595	9.190	6.265	4.284	2.865
copies = 3 (MARS)	22.064	14.917	9.719	7.147	5.450

Curare (stop&copy)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	19.725	18.315	16.241	14.546	12.912
copies = 1 (DIS)	15.034	13.636	11.633	9.906	8.285
copies = 3 (DIS)	13.421	12.000	9.865	8.217	6.788
copies = 10 (DIS)	11.953	10.245	8.041	6.470	3.845
copies = 3 (MARS)	10.527	8.605	7.489	5.560	3.756

Curare (mark&sweep)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	19.725	18.315	16.241	14.546	12.912
copies = 1 (DIS)	17.404	16.235	14.168	12.447	10.869
copies = 3 (DIS)	16.019	14.476	12.250	10.710	9.008
copies = 10 (DIS)	14.082	12.726	10.707	9.047	6.878
copies = 3 (MARS)	13.200	11.439	9.891	6.305	3.355

RL (stop&copy)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	17.867	15.887	13.204	10.748	8.531
copies = 1 (DIS)	14.384	11.026	8.474	6.451	4.523
copies = 3 (DIS)	11.354	8.531	6.508	4.340	2.656
copies = 10 (DIS)	8.729	6.554	4.205	2.234	1.157
copies = 3 (MARS)	21.052	11.598	7.946	5.921	3.661

RL (mark&sweep)

Copy Number	Promotion Rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
copies = 0 (DIS)	17.867	15.887	13.204	10.748	8.531
copies = 1 (DIS)	16.096	13.453	10.921	8.557	6.553
copies = 3 (DIS)	13.993	11.312	8.976	6.772	4.770
copies = 10 (DIS)	11.519	8.965	6.397	4.366	2.732
copies = 3 (MARS)	24.003	18.830	12.151	6.668	5.554

Table D.25: Promotion Rates for Three Applications

Lisp Compiler

Newspace	Collection Frequency (min)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.878	2.094	4.512	9.362	19.070
5%	0.169	0.520	1.484	3.422	7.302
10%	0.097	0.229	0.658	1.627	3.568
20%	0.044	0.099	0.234	0.668	1.638

Curare

Newspace	Collection Frequency (min)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	2.585	5.898	11.237	21.506	35.676
5%	0.425	1.941	3.883	7.974	17.844
10%	0.161	0.386	1.429	3.882	7.973
20%	0.078	0.162	0.385	1.391	3.882

Boyer Moore TP

Newspace	Collection Frequency (min)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	3.881	9.346	20.165	40.410	86.019
5%	1.104	3.224	7.564	16.157	33.858
10%	0.422	1.107	3.226	7.564	16.164
20%	0.186	0.423	1.112	3.233	7.564

RL

Newspace	Collection Frequency (min)				
Promotion	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.428	1.429	3.636	8.280	17.400
5%	0.188	0.468	1.301	3.128	6.778
10%	0.094	0.242	0.602	1.487	3.313
20%	0.039	0.095	0.245	0.603	1.493

Table D.26: Second Generation Collection Frequencies for Four Applications

Lisp Compiler

Newspace Promotion	Pause Length (sec)				
	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.286	0.285	0.290	0.298	0.307
5%	0.797	0.970	0.974	0.983	0.990
10%	0.762	1.231	1.333	1.341	1.351
20%	0.820	1.498	2.397	2.581	2.592

Curare

Newspace Promotion	Pause Length (sec)				
	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.153	0.051	0.167	0.230	0.690
5%	0.756	0.431	0.874	1.099	0.750
10%	0.975	1.753	1.699	1.906	2.420
20%	1.011	1.954	3.512	3.518	3.450

Boyer Moore TP

Newspace Promotion	Pause Length (sec)				
	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.295	0.298	0.298	0.312	0.311
5%	0.529	0.534	0.540	0.545	0.554
10%	0.761	1.055	1.065	1.077	1.087
20%	0.840	1.517	2.098	2.116	2.164

RL

Newspace Promotion	Pause Length (sec)				
	$th = 500$	$th = 1000$	$th = 2000$	$th = 4000$	$th = 8000$
2%	0.716	0.772	0.841	0.783	0.797
5%	0.713	1.115	1.192	1.190	1.200
10%	0.707	1.109	1.471	1.536	1.543
20%	0.826	1.405	2.188	2.934	3.035

Table D.27: Second Generation Pause Lengths for Four Applications

Lisp Compiler (stop&copy)

Threshold (kbytes)	Pause Length (msec)		
	copies=0	copies=3	copies=10
2000	3.502	6.988	10.921
4000	5.173	10.447	12.493
8000	7.869	12.483	12.902
16000	11.409	13.099	13.485
32000	13.095	13.801	14.570

RL (stop&copy)

Pause Length (msec)		
copies=0	copies=3	copies=10
3.535	7.031	10.264
5.436	9.909	14.262
7.855	13.518	15.886
10.692	16.018	16.652
14.720	17.081	17.871

Lisp Compiler (stop&copy)

Threshold (kbytes)	Collection Frequency (sec)		
	copies=0	copies=3	copies=10
2000	0.058	0.050	0.043
4000	0.117	0.104	0.099
8000	0.233	0.216	0.215
16000	0.466	0.448	0.448
32000	0.932	0.913	0.912

RL (stop&copy)

Collection Frequency (sec)		
copies=0	copies=3	copies=10
0.056	0.048	0.043
0.109	0.098	0.091
0.219	0.203	0.198
0.438	0.418	0.417
0.877	0.854	0.854

Lisp Compiler (stop&copy)

Threshold (kbytes)	Promotion Rate (%)		
	copies=0	copies=3	copies=10
2000	8.496	3.011	1.289
4000	6.286	1.864	0.069
8000	4.776	0.156	0.000
16000	3.464	0.000	0.000
32000	1.988	0.000	0.000

RL (stop&copy)

Promotion Rate (%)		
copies=0	copies=3	copies=10
8.531	2.656	1.157
6.577	1.444	0.588
4.753	0.949	0.033
3.235	0.098	0.000
2.227	0.001	0.000

Table D.28: Predicted Performance for Two Applications. Predicted CPU speed is 100 times a Sun4/280.

Lisp Compiler (stop&copy)

Threshold (kbytes)	Pause Length (msec)		
	copies=0	copies=3	copies=10
2000	17.112	28.094	34.687
4000	30.239	51.998	66.471
8000	52.628	93.461	122.555
16000	89.200	150.755	202.063
32000	138.338	225.222	325.405
64000	193.839	340.472	523.867
128000	273.978	521.810	842.094

RL (stop&copy)

Pause Length (msec)		
copies=0	copies=3	copies=10
8.533	20.018	29.258
16.416	38.361	56.406
31.128	70.639	104.575
57.040	122.699	187.373
98.367	208.880	326.742
160.814	344.673	537.609
261.586	539.946	816.351

Lisp Compiler (stop&copy)

Threshold (kbytes)	Collection Frequency (sec)		
	copies=0	copies=3	copies=10
2000	0.058	0.027	0.013
4000	0.117	0.058	0.029
8000	0.233	0.126	0.069
16000	0.466	0.281	0.193
32000	0.932	0.655	0.495
64000	1.864	1.451	1.165
128000	3.728	3.103	2.604

RL (stop&copy)

Collection Frequency (sec)		
copies=0	copies=3	copies=10
0.056	0.035	0.019
0.109	0.070	0.040
0.219	0.145	0.089
0.438	0.305	0.205
0.877	0.649	0.468
1.754	1.372	1.078
3.508	2.897	2.471

Lisp Compiler (stop&copy)

Threshold (kbytes)	Promotion Rate (%)		
	copies=0	copies=3	copies=10
2000	41.560	30.819	28.118
4000	36.774	25.737	22.724
8000	31.962	19.719	14.145
16000	27.095	9.948	6.359
32000	21.007	6.149	4.462
64000	14.722	4.616	3.312
128000	10.406	3.555	2.296

RL (stop&copy)

Promotion Rate (%)		
copies=0	copies=3	copies=10
20.614	18.004	16.651
19.874	16.390	14.035
18.845	13.350	10.108
17.265	9.604	7.597
14.884	7.685	5.329
12.170	5.550	3.333
9.898	3.661	1.777

Table D.29: Predicted Performance for Two Applications. Predicted lifespans are 100 times those measured. Predicted CPU speed is 100 times a Sun4/280.

Lisp Compiler (stop&copy)

Threshold (kbytes)	Pause Length (msec) pr=10%	pr=20%	pr=30%	pr=40%
2000	39.49	37.93	38.59	37.15
4000	76.17	67.94	72.78	70.40
8000	146.11	122.43	125.25	124.88
16000	275.07	217.12	209.39	199.83
32000	509.78	378.72	347.57	321.31
64000	908.59	654.02	573.52	516.55
128000	1242.42	1028.82	921.77	829.21
256000	1377.37	1146.88	1183.41	1179.80
512000	1309.04	1250.22	1236.33	1243.67
1024000	1313.05	1197.63	1311.46	1361.32

RL (stop&copy)

Pause Length (msec) pr=10%	pr=20%	pr=30%	pr=40%
40.19	38.88	38.96	39.26
78.66	73.71	75.10	76.40
151.50	135.17	139.62	144.25
282.90	247.28	253.94	263.32
503.99	429.38	452.31	477.66
823.90	694.25	760.59	820.33
1263.72	1020.39	1173.85	1314.82
1484.33	1426.86	1674.33	1914.14
1518.19	1527.26	2108.29	2649.45
1551.15	1558.08	2149.52	2830.66

Lisp Compiler (stop&copy)

Threshold (kbytes)	Collection Frequency (sec) pr=10%	pr=20%	pr=30%	pr=40%
2000	0.15	0.08	0.05	0.04
4000	0.32	0.18	0.11	0.09
8000	0.70	0.45	0.27	0.21
16000	1.55	1.06	0.71	0.55
32000	3.51	2.44	1.72	1.36
64000	7.97	5.50	3.95	3.12
128000	20.34	12.27	8.73	6.85
256000	51.58	29.53	19.46	14.82
512000	129.16	61.40	43.43	33.18
1024000	278.31	139.15	88.05	65.10

RL (stop&copy)

Collection Frequency (sec) pr=10%	pr=20%	pr=30%	pr=40%
0.14	0.07	0.05	0.04
0.29	0.16	0.11	0.08
0.61	0.36	0.23	0.17
1.36	0.85	0.54	0.38
3.19	1.99	1.26	0.89
7.97	4.77	2.98	2.09
20.42	11.49	7.15	4.97
51.17	26.56	16.94	11.87
120.11	60.59	37.58	27.18
256.68	130.03	84.75	61.36

Lisp Compiler (stop&copy)

Threshold (kbytes)	Promotion Rate (%) pr=10%	pr=20%	pr=30%	pr=40%
2000	85.10	72.04	77.52	67.20
4000	73.42	44.40	60.90	54.90
8000	62.63	33.01	30.96	33.98
16000	50.56	24.93	19.99	16.84
32000	40.05	18.78	14.60	11.91
64000	27.49	14.53	11.06	8.96
128000	2.28	7.46	7.34	6.60
256000	0.01	0.39	0.90	2.55
512000	0.00	0.00	0.01	0.12
1024000	0.00	0.00	0.00	0.00

RL (stop&copy)

Promotion Rate (%) pr=10%	pr=20%	pr=30%	pr=40%
91.25	79.82	80.89	83.23
83.75	64.07	69.79	74.75
71.84	46.11	52.02	59.37
54.73	35.42	38.27	42.34
35.45	23.34	27.14	32.13
17.76	12.96	17.02	20.77
10.12	6.39	9.00	11.53
0.58	3.33	5.11	5.82
0.00	0.17	0.78	2.93
0.00	0.00	0.00	0.15

Table D.30: Second Generation Metrics for Four Applications Assuming Longer Running Programs and Faster CPU's. Object lifespan is assumed to be 100 times the lifespan actually observed.

Lisp Compiler (stop&copy)

Threshold Size (kbytes)	Pause Length (msec) pr = 2%	pr = 4%	pr = 6%	pr = 8%
2000	40.44	40.74	39.92	39.22
4000	79.60	80.88	78.21	76.72
8000	153.41	159.18	151.84	146.71
16000	277.97	308.07	290.11	276.81
32000	374.70	569.71	544.29	516.87
64000	377.33	844.50	920.17	901.55
128000	325.20	865.11	1000.10	1153.29
256000	423.32	746.03	1079.48	1148.75
512000	417.14	968.46	1161.26	995.27
1024000	377.19	952.33	1014.64	1295.91

RL (stop&copy)

Pause Length (msec) pr = 2%	pr = 4%	pr = 6%	pr = 8%
40.30	40.41	40.37	40.22
78.98	79.48	79.31	78.64
150.41	152.78	152.03	151.00
278.50	287.08	287.20	281.13
509.90	506.60	510.40	499.70
675.34	881.40	849.09	814.31
637.97	1076.78	1214.30	1245.20
640.24	1008.02	1316.08	1352.16
664.46	1013.23	1453.13	1264.84
924.07	1055.42	1155.45	1273.45

Lisp Compiler (stop&copy)

Threshold Size (kbytes)	Collection Frequency (sec) pr = 2%	pr = 4%	pr = 6%	pr = 8%
2000	0.74	0.37	0.25	0.20
4000	1.52	0.75	0.52	0.41
8000	3.17	1.52	1.10	0.87
16000	7.06	3.17	2.38	1.91
32000	20.27	6.93	5.23	4.29
64000	65.67	17.31	12.22	9.85
128000	188.09	61.73	39.90	25.86
256000	305.94	183.93	94.95	71.93
512000	658.31	302.24	198.43	194.09
1024000	1543.82	654.61	492.39	311.49

RL (stop&copy)

Collection Frequency (sec) pr = 2%	pr = 4%	pr = 6%	pr = 8%
0.70	0.35	0.23	0.18
1.44	0.72	0.48	0.36
3.06	1.50	1.01	0.77
6.95	3.32	2.21	1.70
16.07	7.74	5.21	3.99
44.55	19.13	13.21	10.05
145.43	53.25	33.65	25.51
337.17	155.04	85.60	66.35
702.74	346.42	178.90	168.67
1045.26	712.82	504.67	359.86

Table D.31: Third Generation Metrics for Four Applications Assuming Longer Running Programs and Faster CPU's. Object lifespan is assumed to be 100 times the lifespan actually observed.

Lisp Compiler

Cache Size	Total miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (512k)	0.390	0.463	0.472	0.487	0.477
mark-and-sweep (512k)	0.194	0.226	0.358	0.462	0.487
stop-and-copy (1M)	0.326	0.379	0.362	0.409	0.407
mark-and-sweep (1M)	0.125	0.133	0.231	0.319	0.406
stop-and-copy (2M)	0.123	0.141	0.111	0.167	0.361
mark-and-sweep (2M)	0.105	0.111	0.181	0.236	0.305

Curare

Cache Size	Total miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (512k)	0.167	0.157	0.204	0.176	0.174
mark-and-sweep (512k)	0.198	0.158	0.166	0.171	0.224
stop-and-copy (1M)	0.082	0.122	0.133	0.099	0.090
mark-and-sweep (1M)	0.151	0.116	0.072	0.082	0.107
stop-and-copy (2M)	0.043	0.069	0.079	0.069	0.057
mark-and-sweep (2M)	0.092	0.047	0.057	0.066	0.062

Boyer Moore TP

Cache Size	Total miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (512k)	0.142	0.141	0.152	0.161	0.152
mark-and-sweep (512k)	0.099	0.109	0.127	0.147	0.155
stop-and-copy (1M)	0.097	0.104	0.106	0.113	0.113
mark-and-sweep (1M)	0.059	0.067	0.072	0.082	0.114
stop-and-copy (2M)	0.053	0.052	0.059	0.083	0.093
mark-and-sweep (2M)	0.044	0.052	0.056	0.060	0.087

RL

Cache Size	Total miss rate (%)				
	$th = 125$	$th = 250$	$th = 500$	$th = 1000$	$th = 2000$
stop-and-copy (512k)	0.489	0.465	0.572	0.548	0.577
mark-and-sweep (512k)	0.536	0.464	0.460	0.426	0.412
stop-and-copy (1M)	0.398	0.384	0.422	0.422	0.444
mark-and-sweep (1M)	0.397	0.293	0.263	0.251	0.299
stop-and-copy (2M)	0.151	0.166	0.216	0.248	0.348
mark-and-sweep (2M)	0.246	0.226	0.215	0.208	0.187

Table D.32: Total Cache Miss Rates for Three Collection Algorithms.

Ifetch Miss Rates

Cache Size	Instruction cache miss rate (%)			
(kbytes)	SLC	RSIM	Weaver	PMA
32	2.21	1.34	1.72	2.40
64	1.54	0.87	0.53	1.55
128	0.92	0.85	0.49	1.08
256	0.52	0.03	0.01	0.54
512	0.21	0.03	0.01	0.21
1024	0.11	0.03	0.00	0.11
2048	0.06	0.03	0.00	0.08

Table D.33: Instruction Cache Miss Rates for Four Applications.

Speedup

Actual	Effective Uniprocessors				
Processors	0.1%	0.3%	0.5%	1%	1.5%
1	1.00	1.00	1.00	1.00	1.00
5	5.00	4.98	4.94	4.76	4.49
10	9.99	9.88	9.63	8.35	6.74
20	19.94	19.24	17.00	10.08	7.06
30	29.85	26.99	19.13	10.09	7.06
40	39.70	30.80	19.18	10.09	7.06
50	49.43	31.29	19.18	10.09	7.06
60	58.97	31.30	19.18	10.09	7.06
70	68.18	31.30	19.18	10.09	7.06
80	76.76	31.30	19.18	10.09	7.06
90	84.06	31.30	19.18	10.09	7.06
100	89.11	31.30	19.18	10.09	7.06

Table D.34: Maximum Effective Uniprocessors for Different Miss Ratios.

Lisp Compiler (stop&copy)

Actual Processors	Effective Uniprocessors		
	$th = 125$	$th = 500$	$th = 2000$
1	0.64	0.81	0.87
5	3.17	4.02	4.31
10	6.28	7.96	8.48
20	12.13	15.18	15.86
30	16.62	20.00	19.78
40	18.27	21.10	20.24
50	18.39	21.13	20.24
60	18.39	21.13	20.24
70	18.39	21.13	20.24
80	18.39	21.13	20.24
90	18.39	21.13	20.24
100	18.39	21.13	20.24

Lisp Compiler (mark&sweep)

Effective Uniprocessors		
$th = 125$	$th = 500$	$th = 2000$
0.63	0.79	0.87
3.15	3.95	4.32
6.30	7.86	8.50
12.56	15.52	15.91
18.76	22.69	19.87
24.88	28.50	20.34
30.83	31.42	20.35
36.44	31.90	20.35
41.34	31.92	20.35
44.82	31.92	20.35
46.36	31.92	20.35
46.66	31.92	20.35

Curare (stop&copy)

Actual Processors	Effective Uniprocessors		
	$th = 125$	$th = 500$	$th = 2000$
1	0.87	0.91	0.94
5	4.33	4.56	4.69
10	8.66	9.12	9.37
20	17.31	18.18	18.72
30	25.93	27.15	28.03
40	34.52	35.95	37.30
50	43.06	44.43	46.50
60	51.53	52.24	55.58
70	59.89	58.58	64.49
80	68.08	62.30	73.07
90	75.95	63.41	81.05
100	83.26	63.53	87.87

Curare (mark&sweep)

Effective Uniprocessors		
$th = 125$	$th = 500$	$th = 2000$
0.86	0.89	0.93
4.28	4.44	4.64
8.54	8.88	9.27
17.02	17.74	18.51
25.36	26.58	27.70
33.45	35.40	36.81
41.00	44.19	45.78
47.31	52.94	54.52
51.18	61.63	62.81
52.37	70.22	70.19
52.49	78.67	75.81
52.50	86.88	78.79

RL (stop&copy)

Actual Processors	Effective Uniprocessors		
	$th = 125$	$th = 500$	$th = 2000$
1	0.58	0.74	0.85
5	2.86	3.65	4.20
10	5.63	7.19	8.25
20	10.59	13.34	15.12
30	13.36	16.33	18.03
40	13.74	16.61	18.22
50	13.75	16.62	18.22
60	13.75	16.62	18.22
70	13.75	16.62	18.22
80	13.75	16.62	18.22
90	13.75	16.62	18.22
100	13.75	16.62	18.22

RL (mark&sweep)

Effective Uniprocessors		
$th = 125$	$th = 500$	$th = 2000$
0.58	0.72	0.84
2.86	3.57	4.18
5.64	7.11	8.30
10.59	13.95	16.16
13.39	20.08	22.69
13.78	24.23	25.92
13.78	25.43	26.35
13.78	25.50	26.36
13.78	25.50	26.36
13.78	25.50	26.36
13.78	25.50	26.36
13.78	25.50	26.36

Table D.35: Maximum Effective Uniprocessors for Different Algorithms and Threshold Sizes. Garbage collection induced overhead is taken into account.

Bibliography

[1] D. Allen, S. Steinberg, and L. Stabile. Recent developments in Butterfly Lisp. In *Proc. AAAI-87 Sixth National Conference on Artificial Intelligence*, July 1987.

[2] Andrew Appel, John Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, GA, June 1988. SIGPLAN, ACM Press.

[3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–183, February 1989.

[4] Stefan Arnborg. Optimal memory management in a system with garbage collection. *BIT*, 14:375–381, 1974.

[5] H. D. Baecker. Garbage collection for virtual memory computer systems. *Communications of the ACM*, 15(11):981–986, November 1972.

[6] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[7] Mary Gray Baker. Discussion about sprite protection faults. Personal communication., October 1989.

[8] David R. Barach, David H. Taenzer, and Robert E. Wells. A technique for finding storage allocation errors in C-language programs. *ACM SIGPLAN Notices*, 17(5):16–23, March 1982.

[9] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, pages 353–357, July 1975.

[10] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe*, pages 273–288. Springer-Verlag, June 1987. LNCS 259.

[11] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, MIT Laboratory for Computer Science, Cambridge, Massachusetts, May 1977.

[12] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In Norman Meyrowitz, editor, *OOPSLA'86 Conference Proceedings*, pages 17–37, Portland, OR, November 1986. ACM.

[13] Daniel G. Bobrow and Daniel L. Murphy. Structure of a Lisp system using two-level storage. *Communications of the ACM*, 10(3):155–159, March 1967.

[14] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press, Inc., Boston, 1988.

[15] D. R. Brownbridge. *Recursive Structures in Computer Systems*. PhD thesis, University of Newcastle on Tyne, September 1984.

[16] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 report. Technical Report Research Report 31, Digital Equipment Corporation System Research Center, Palo Alto, CA, August 1988.

[17] Patrick Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In Normam Meyrowitz, editor, *OOPSLA'86 Conference Proceedings*, pages 119–130, Portland, OR, September 1986. ACM.

[18] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[19] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–87, February 1977.

[20] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, 1981.

[21] Jacques Cohen. A use of fast and slow memories in list-processing languages. *Communications of the ACM*, 10(2):82–86, February 1967.

[22] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, October 1983.

[23] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[24] Anthony James Cortemanche. MultiTrash, a parallel garbage collector for multiScheme. Bachelor's thesis, MIT, January 1986.

[25] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[26] D. Julian M. Davies. Memory occupancy patterns in garbage collection systems. *Communications of the ACM*, 27(8):819–825, August 1984.

[27] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[28] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[29] John R. Ellis, Kai Li, and Andrew Appel. Real-time concurrent collection on stock multiprocessors. Technical Report 25, DEC Systems Research Center, Palo Alto, CA, February 1988.

[30] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage-collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[31] John Foderaro, Keith Sklower, Kevin Layer, et al. *Franz Lisp Reference Manual.* Franz Inc., Berkeley, CA, 1985.

[32] John K. Foderaro and Richard J. Fateman. Characterization of VAX Macsyma. In *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Manipulation*, pages 14–19. ACM, 1981.

[33] Franz Incorporated. *Allegro Common Lisp User Guide*, Release 3.0 (beta) edition, April 1988.

[34] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press Series in Computer Science. M.I.T. Press, Cambridge, MA, 1985.

[35] Richard P. Gabriel. *Performance Evaluation of Lisp Systems.* Computer Systems Series. MIT Press, Cambridge, Massachusetts, 1985.

[36] Richard P. Gabriel and John McCarthy. Qlisp. In Janusz S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 63–90. Kluwer Academic Publishers, 1988.

[37] Adele Goldberg and David Robson. *Smalltalk-80: the Language and Its Implementation.* Series in Computer Science. Addison-Wesley, Palo Alto, CA, 1983.

[38] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 313–320, Portland, OR, June 1989.

[39] Ron Goldman and Richard P. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, 6(4):51–59, July 1989.

[40] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[41] Tim Hickey and Jacques Cohen. Performance analysis of on-the-fly garbage collection. *Communications of the ACM*, 27(11):1143–1154, November 1984.

[42] Mark Hill, Susan Eggers, James Larus, George Taylor, et al. SPUR: A VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–22, November 1986.

[43] Mark D. Hill. *TYCHO*. University of Wisconsin, Madison, WI. Unix manual page.

[44] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California at Berkeley, Berkeley, CA, November 1987. Also appears as tech report UCB/CSD 87/381.

[45] C. A. R. Hoare. Optimization of store size for garbage collection. *Information Processing Letters*, 2:165–166, February 1974.

[46] Motokuza Hozumi, Toshiaki Kurokawa, Norihisa Suzuki, Tomoyuki Tanaka, and Shigeru Uzuhara. Multiprocessor Common Lisp on TOP-1. In *US/Japan Workshop on Parallel Lisp*, Sendai, Japan, June 1989.

[47] Douglas Johnson. Trap architectures for Lisp systems. Technical Report UCB/CSD 88/470, UCBCS, Berkeley, CA, November 1988.

[48] Douglas Johnson. Discussion about a large lisp cad system. Personal communication., October 1989.

[49] Mike Johnson. *Am29000 User's Manual*. Advanced Micro Devices, 1987.

[50] Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[51] David Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 81–90, Portland, OR, June 1989. SIGPLAN, ACM Press.

[52] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. Technical note, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1977.

[53] Leslie Lamport. Garbage collection with multiple processes: An exercise in parallelism. In *Proceedings of the International Conference on Parallel Processing*, pages 50–54, Walden Woods, Massachusetts, August 1976. IEEE.

[54] James R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California, Berkeley, Berkeley, CA, 1989.

[55] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[56] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Shaffert, R. Scheifler, and A. Snyder. *The CLU Reference Manual.* Lecture Notes in Computer Science. Springer-Verlag, New York, New York, 1981.

[57] LMI. The lambda system: Technical summary. Technical report, LISP Machines, Inc., 1983.

[58] Shogo Matsui, Yoshinobu Kato, Shinsuke Teramura, Tomoyuki Tanaka, Nobuyuki Mohri, Atsushi Maeda, and Masakazu Nakanishi. Synapse: A multi-microprocessor Lisp machine with parallel garbage collector. In *Proceedings of the International Workshop on Parallel Algorithms and Architectures*, pages 131–137, Suhl, GDR, March 1987.

[59] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[60] John McCarthy. Recursive functions of symbolic expressions and their computations by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[61] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[62] David A. Moon. Architecture of the Symbolics 3600. In *Proceedings of the Twelfth Symposium on Computer Architecture*, Boston, Massachusetts, June 1985.

[63] Mike N. Nelson. *Physical Memory Management in a Network Operating System.* PhD thesis, University of California at Berkeley, Berkeley, CA, November 1988. Also appears as tech report UCB/CSD 88/471.

[64] I. A. Newman, R. P. Stallard, and M. C. Woodward. Improved multiprocessor garbage collection algorithms. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 367–368, Ohio State University, Columbus, OH, August 1983. IEEE.

[65] I. A. Newman and M. C. Woodward. Alternative approaches to multiprocessor garbage collection. In *Proceedings of the 1982 International Conference on Parallel Processing*, pages 205–210, Ohio State University, Columbus, OH, August 1982. IEEE.

[66] Peter Nuth and Robert Halstead, Jr. A study of LISP on a multiprocessor (preliminary version). *Lisp Pointers*, 2(3–4):15–32, 1989.

[67] Frank Olken. Efficient methods for calculating the success function of fixed space replacement policies. Master's thesis, University of California at Berkeley, Berkeley, CA, March 1981.

[68] C.-J. Peng and G. S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Dept., Univ. of Wisconsin—Madison, July 1989.

[69] R. Rashid et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Second Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 31–39. ACM, October 1987.

[70] Ken Rimey and Paul N. Hilfinger. A compiler for application-specific signal processors. In *VLSI Signal Processing, III*, pages 341–351. IEEE Press, November 1988.

[71] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, Palo Alto, California, July 1985.

[72] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[73] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University, March 1987.

[74] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Stanford, CA, February 1988. Also appears as Computer Systems Laboratory tech report CSL-TR-88-351.

[75] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general purpose computers. Bachelor's thesis, MIT, 1988.

[76] Guy L. Steele, Jr. *Common Lisp: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[77] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[78] Peter Steenkiste. Lisp on a reduced-instruction-set processor: Characterization and optimization. Technical Report CSL-TR-87-324, Computer Systems Laboratory, Stanford University, March 1987. PhD dissertation.

[79] Sun Microsystems, Inc., Mountain View, CA. *The SPARC Architecture Manual*, revision 50 edition, August 1987.

[80] George Taylor. Ratio of MIPS R3000 instructions to heap references. Personal communication, October 1989.

[81] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp architecture. In *Proceedings of the Thirteenth Symposium on Computer Architecture*, June 1986.

[82] Chris J. Terman. Simulation tools for digital LSI design. Technical Report TR-304, MIT Laboratory for Computer Science, Cambridge, MA, September 1983.

[83] James G. Thompson. *Efficient Analysis of Caching Systems.* PhD thesis, University of California at Berkeley, Berkeley, CA, October 1987. Also appears as tech report UCB/CSD 87/374.

[84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[85] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *OOPSLA'88 Conference Proceedings*, pages 1–17. ACM, September 1988.

[86] David M. Ungar. *The Design and Evaluation of A High Performance Smalltalk System.* PhD thesis, University of California at Berkeley, Berkeley, CA, March 1986. Also appears as tech report UCB/CSD 86/287.

[87] Philip L. Wadler. Analysis of an algorithm for real time garbage collection. *Communications of the ACM*, 19(9):491–500, September 1976.

[88] David W. Wall. Global register allocation at link time. Technical Report 86/3, DEC Western Research Laboratory, Palo Alto, CA, October 1986.

[89] Skef Wholey, Scott Fahlman, and Joseph Ginder. Revised internal design of Spice Lisp. Technical report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, January 1985.

[90] Paul R. Wilson. Opportunistic garbage collection. *SIGPLAN Notices*, 23(12):98–102, December 1988.

[91] Paul R. Wilson and Thomas G. Moher. Design of an efficient generation garbage collector. To appear., April 1989.

[92] Taiichi Yuasa. Realtime garbage collection on general-purpose machines. Technical Report Preprint 535, Research Institute for Mathematical Science, Kyoto University, Japan, 1986.

[93] Taiichi Yuasa and Masami Hagiya. *The KCL Report.* Research Institute for Mathematical Sciences, University of Kyoto.

[94] Benjamin Zorn and Paul Hilfinger. Direct function calls in SPUR Lisp. Technical Report UCB/CSD 88/403, Computer Science Division (EECS), University of California, Berkeley, February 1988.

[95] Benjamin Zorn and Paul Hilfinger. A memory allocation profiler for C and Lisp programs. In *Proceedings of the Summer 1988 USENIX Conference*, San Francisco, CA, June 1988.

[96] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. SPUR Lisp: Design and implementation. Technical Report UCB/CSD 87/373, Computer Science Division (EECS), University of California, Berkeley, October 1987.

[97] Benjamin Zorn, Kinson Ho, James Larus, Luigi Semenzato, and Paul Hilfinger. Multiprocessing extensions in SPUR Lisp. *IEEE Software*, 6(4):41–49, July 1989.